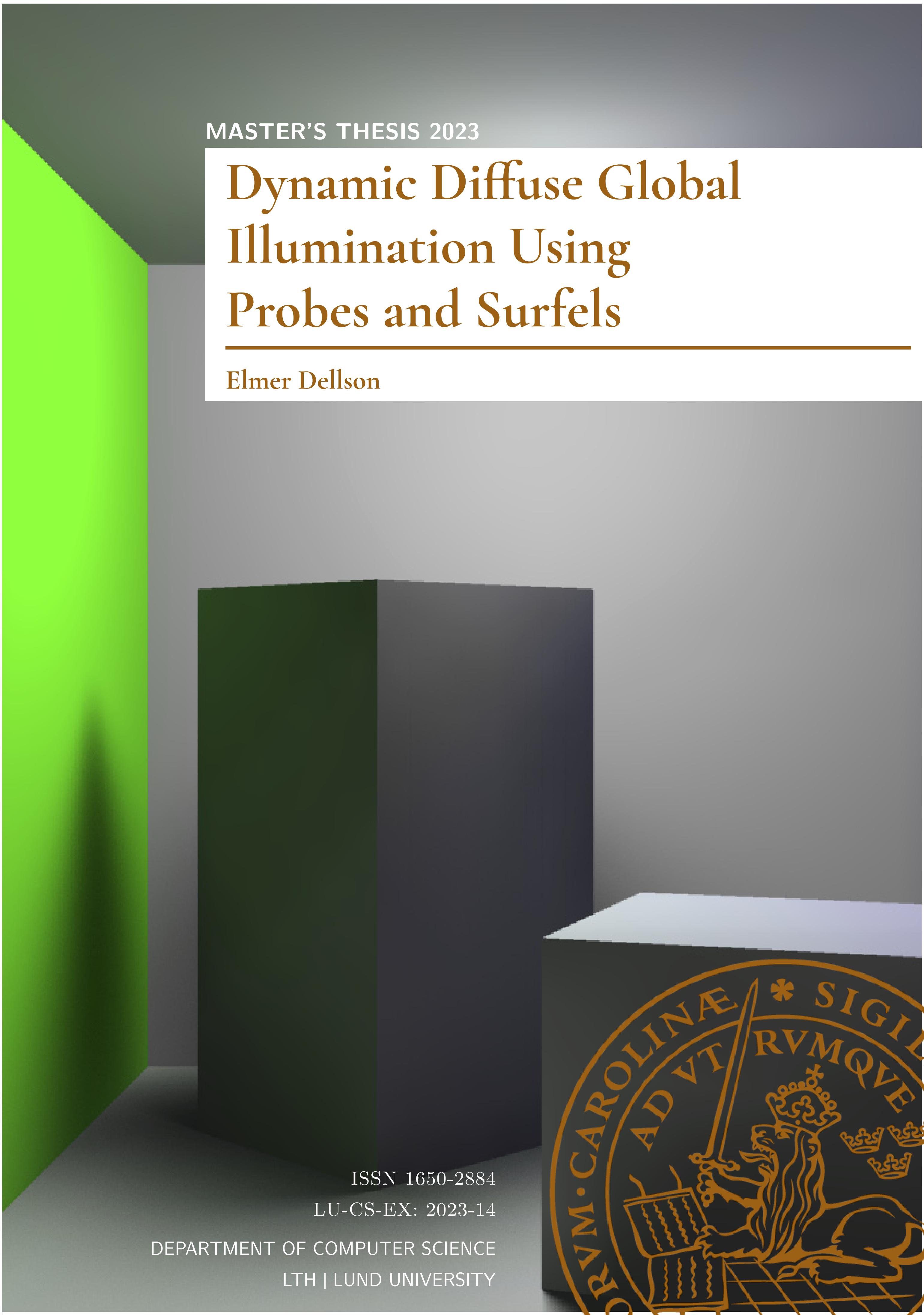


MASTER'S THESIS 2023

Dynamic Diffuse Global Illumination Using Probes and Surfels

Elmer Dellson

ISSN 1650-2884

LU-CS-EX: 2023-14

DEPARTMENT OF COMPUTER SCIENCE

LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-14

Dynamic Diffuse Global Illumination Using Probes and Surfels

Dynamisk lambertsk global belysning genom sonder och ytelement

Elmer Dellson

Dynamic Diffuse Global Illumination Using Probes and Surfels

Elmer Dellson
`elmer@dellson.com`

June 28, 2023

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Michael Doggett (LTH), `michael.doggett@cs.lth.se`
Calle Lejdfors (Tencent Games, AMD), `calle.lejdfors@amd.com`

Examiner: Per Andersson, `per.andersson@cs.lth.se`

Abstract

One of the major challenges of real-time computer graphics is that of Global Illumination (GI). This thesis explores an approach to dynamic diffuse GI based on light probes and surfels combined with hardware accelerated ray tracing to achieve convincing bounce-light illumination in 3D scenes with real-time frame rates. The system places surfels from the probes using ray tracing, that are then used to cache the geometry information of the surface they are placed on. This information is used to light the surfels, which feed the lighting data into the probes. The probes then propagate the light from the surfels across the scene, and use it to add bounce lighting to the geometry. The system is fully dynamic and does not require any pre-baking, so that all geometry in the scene can be moved while the application is running.

The results show that the technique yields good bounce lighting for the simple scenes sampled. We also saw that the caching of surfels saves greatly on the computational cost of the system compared to ray tracing to replace them every frame, decreasing the time spent by $\sim 85\%$.

Keywords: Graphics, Global Illumination, Real-Time, Ray Tracing

Acknowledgments

I want to thank Calle Lejdfors and Michael Doggett for their help and guidance along the way.

I also want to thank Martin-Karl Lefrançois and Pascal Gautron for the DX12 ray tracing tutorial that got me started.

Finally, I want to thank Matilda for her support and encouragement.

Contents

Chapter 1

Introduction

One of the big problems that need solving for real-time 3D graphics applications is that of *Global Illumination* (GI). GI describes the way that light spreads around a scene and bounces between objects, having them illuminate each other through indirect bounced light. This is a key aspect of realistic lighting in computer graphics, since bounced light is an important part of how we perceive the real world. However, accurately describing how light behaves is a computationally difficult problem: billions of photons bounce around a room every second at literally the speed of light, changing their properties with every bounce, until they dissipate or finally hit your eye to produce an image. To avoid doing the immense amount of maths required to accurately simulate light in the real world, we want to figure out clever ways to produce images that look as true-to-life as possible, while still being fast enough that the system can be interacted with in real time.

We designed a system to achieve convincing GI using *light probes*, which is a type of structure that stores lighting information in 3D space. This is combined with *surfels* (short for *surface elements*) to discretize and cache the sampling of geometry and reduce the amount of lighting calculations needed. To place the surfels, and to render the final image, we used hardware-accelerated *ray tracing*.

While previous implementations of similar schemes have relied on pre-calculating the light transport data for a fixed scene, meaning that most of the geometry cannot move while the application is running, we implemented our solution in a way that supports dynamic changes to all parts of the scene.

1.1 Research Questions

The research questions for this thesis are:

- Is a combination of light probes and surfels together with ray tracing a good approach to achieve convincing global illumination in a dynamic scene for real-time 3D applications?

- How much performance is gained by using surfels compared to not using them?

1.2 Project Scope

In order to investigate the questions posed above, we built a simple renderer in the DirectX 12 API, and implemented a dynamic diffuse GI system based on probes and surfels into it.

To limit the scope of this project suitably for a Master's thesis, we implemented our solution in small test scenes with very simple geometry, and the following limitations:

- We only cared about second order light contribution.
- We did not investigate how best to place the probes; instead, we just placed them in a regular grid.
- We did not implement any advanced material rendering. All surfaces were considered purely diffuse and opaque, with no textures or normal maps, etc. applied.

1.3 Contribution to the State of Knowledge

This thesis contributes new knowledge about the possibilities of dynamic diffuse global illumination through probes and surfels together with ray tracing, allowing future renderers to either implement the techniques discovered or explore other avenues having learned from the limitations found.

1.4 Source Code

The full source code of the project is publicly available at:
`https://github.com/ElmerDellson/MastersThesis.git`.

Chapter 2
Background and Theory

This chapter contains useful background and theory for understanding the rest of the thesis, mostly concerning real-time 3D graphics, and an overview of recent related work. The technical level is aimed at advanced computer science students, though not necessarily with any experience of real-time graphics.

2.1 Theory

The source for this section is the book *Real-Time Rendering (Fourth Edition)* by Akenine-Möller et al. [1], unless stated otherwise.

2.1.1 Real-Time Computer Graphics Basics

Computer graphics have a great number of applications, from entertainment such as video games and special effects in films, to 3D modeling software and medical imaging technology. This thesis is concerned with *real-time* computer graphics, which is a sub-field where the images (called *frames*) rendered by the computer are produced fast enough that a user can interact with the application, e.g. by moving a virtual camera to change the angle of the image, or move a character around in the scene. In order to achieve a smooth video experience for the user, the goal is to produce (at least) 60 *frames per second* (FPS), and so each frame can take at most $\sim$ 16.7 ms to render.

The Problem

When rendering an image in computer graphics, one fundamental question needs to be answered for every pixel on the screen: *What color is this pixel?* There are many ways of coming up with the answer, all with their particular features and limitations. The way it is usually done in production video games at the time of writing is through a technique called *rasterization*, but since we will not be using this method it will not be covered here. This thesis will use a technique called *ray tracing* to find the answer, and the basics of this approach are explained further on in Section 2.1.2 (*Ray Tracing*).

The Graphics Processing Unit (GPU)

Since there are millions of pixels on a typical modern computer screen, the question of what colors they are needs to be answered millions of times to produce a single frame. The speed required to achieve a high enough *frame rate* imposes very high demands on efficiency, and specialized hardware has been developed to solve this problem: the *Graphics Processing Unit* (GPU). A GPU is a highly parallel processor that can run a large number of threads at the same time. GPUs are programmed using an API such as *Vulkan* or *DirectX*, and this project will use the latter. A GPU can do a number of operations that are important to computer graphics, and only some of these can be programmed by the user.

Shaders

The user-defined programs that run on a GPU are called *shader programs*, or simply *shaders*. For each pixel on the screen, at least one shader is invoked to determine its color. This shader can call other shaders, in any number of combinations, to do quite advanced calculations of things like geometry tessellation and texture filtering. The GPU will run many instances of shaders in parallel in order to speed up the rendering process. For historical reasons, all user-defined programs that run on GPUs are called shaders, even if they do not (directly or indirectly) calculate the color (shade) of a pixel.

Compute Shaders

To use a GPU for arbitrary operations that are not necessarily tied to transforming geometry or calculating the color of a pixel, a *compute shader* can be used. These are GPU programs that can run independently of the main graphics calculations, and any number of them can be invoked per frame regardless of what geometry is in the current scene and the resolution of the display. We will be using a compute shader to run our DDGI system.

HLSL

The shader programs in this thesis are written in the *High-Level Shader Language* (HLSL), which is a C-like programming language used for writing the shader code that is run on the GPU. The specification can be found at Microsoft's website [12].

Scene Representation

The 3D scene to be rendered has to be represented in some way that can be interpreted by the GPU. For this project, we used *vertices* (plural form of *vertex*), a very common method. A vertex is a point in 3D space, used to represent one corner of a piece of primitive geometry, usually a triangle. A large number of triangles (represented by an even larger number of vertices) can then be combined to form complex surfaces that in turn form even more complex geometric shapes. A vertex may also contain additional information, such as a color, which can be used for rendering the geometry. The scenes in this project are simple and contain very few triangles compared to scenes in modern video games.

To light a scene properly we also need at light one *light source*. There are different ways to represent these as well, and we will be using a *point light*. This is represented as a point in space, and a color. When lighting a scene, we will look at the point light's location and color to determine what the lighting looks like.

Camera

When rendering a scene, we "view" it through a virtual camera representation. This virtual camera can have a number of different properties and be modeled in many different ways, but it will at least consist of a location, a viewing direction, and an "up" direction. The properties of the virtual camera are used for rendering the scene from different perspectives, orientations, etc.

Color Representation

Color is represented as a triplet of floating point values, representing the intensities of the red, green and blue channel of a pixel color respectively. The values are in the range $[0, \infty)$, where 0 is perfect black. Standard dynamic range monitors can usually only display colors in the range $[0, 1]$. For this project, any values below 0 are represented as black, and any values above 1 are represented as the maximum intensity.

Diffuse Reflectance

While smooth mirror-like surfaces reflect light rays symmetrically around their normal (known as *specular reflectance*), a *diffuse* material reflects all incoming light rays in random directions. This gives a matte appearance to the surface, and has the important property that it appears the same regardless of the direction from which it is viewed. Diffuse reflectance is often referred to as *Lambertian* reflectance, after Johann Heinrich Lambert who introduced the concept. An illustration can be seen in **Figure 2.1**. The DDGI system is only concerned with diffuse lighting, i.e. lighting of diffuse (Lambertian) surfaces.

The Rendering Equation

Introduced by James Kajiya in 1986, the rendering equation describes the light coming from a point $\mathbf{p}$ on a surface, in the outgoing direction $\mathbf{v}$ (the *view direction*), in terms of the light coming to that point, and the material properties of the surface on which it sits. The equation can be written as

$$L_o(\mathbf{p}, \mathbf{v}) = L_e(\mathbf{p}, \mathbf{v}) + \int_{\mathbf{l} \in \Omega} f(\mathbf{l}, \mathbf{v}) L_o(r(\mathbf{p}, \mathbf{l}), -\mathbf{l})(\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l},$$

where $L_o(\mathbf{p}, \mathbf{v})$ is the outgoing radiance (light) from the point $\mathbf{p}$ in the view direction $\mathbf{v}$, $L_e(\mathbf{p}, \mathbf{v})$ is the emitted radiance from the point $\mathbf{p}$ in direction $\mathbf{v}$, Ω is a hemisphere of all the directions above $\mathbf{p}$, $f(\mathbf{l}, \mathbf{v})$ is the *bidirectional reflectance distribution function* (BRDF) evaluated for $\mathbf{v}$ at the current incoming light direction $\mathbf{l}$, $L_o(r(\mathbf{p}, \mathbf{l}), -\mathbf{l})$ is the incoming radiance

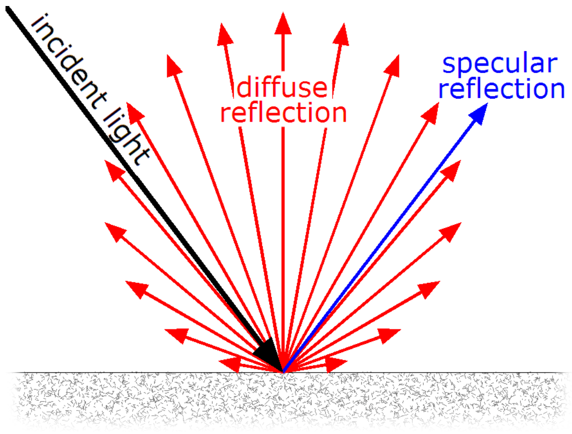


Figure 2.1: Illustration of diffuse reflection [7].

into $\mathbf{p}$ from some other point in the opposite direction to the current incoming light direction $-\mathbf{l}$, and $(\mathbf{n} \cdot \mathbf{l})^{+}$ is the scalar product between $\mathbf{l}$ and the surface normal $\mathbf{n}$ at $\mathbf{p}$ clamped to be non-zero.

This equation sums up the way light behaves in the real world, and if it could be solved fully for every point that lines up with a pixel, we would be done. It is not *a* rendering equation, it is *the* rendering equation (though it can be written in different ways). Unfortunately, the equation is recursive and contains an integral over an infinite number of directions, so we must resort to simplifying it.

Shading Model

In this paper we will be using a common (gross) simplification of the rendering equation to calculate the diffuse lighting at the point $\mathbf{p}$, based on the normal $\mathbf{n}$ of the surface at $\mathbf{p}$, and the incoming direction of the light source $\mathbf{l}$. The base color of a surface as defined by e.g. its vertices or textures, before any lighting or shading is applied to it, is known as its *albedo* color. For a surface with the albedo color $\mathbf{c}_{albedo}$, we get the outgoing color $\mathbf{c}_{out}$ as

$$\mathbf{c}_{out} = \mathbf{c}_{albedo} \cdot max(n \cdot l, 0).$$

Both vectors n and l are normalized to a length of 1, which means that the scalar product $n \cdot l$ between them is also equal to the cosine of the angle between them (since our coordinate system is orthogonal). We take the max with 0 to avoid negative values.

This is a commonly used approximation that takes the relation of the angle between a surface and the light source into account, while still being very fast.

Anti-aliasing

When rendering an image of a mathematical representation of a 3D scene onto a discrete grid of pixels (our computer screen), we will run into some artifacts. One of the most prominent is that of *aliasing*, also known as "jaggies". If we only sample the center of each pixel, we will have no idea how the boundaries between object are changing between the pixel centers, and objects will "pop" from one pixel to another. The left line of **Figure 2.2** shows an example of this: What is supposed to be a straight line comes out as a jagged "staircase". The best solution to this issue would be to increase the resolution of the screen, adding more pixels to smooth the line out, but when that is not possible, the next best thing is to apply some *anti-aliasing* method. There are many to choose from, but we will be employing one of the most straightforward: *Supersampling anti-aliasing* (SSAA). Instead of sampling every pixel only once in its center, we will take four samples per pixel, offset slightly from the pixel center (but still within the boundaries of the pixel), and average them together. The right line in the figure shows what this can look like for a black line moving over the white background: Instead of every pixel being either black or white, we get in-between shades of gray to smooth the transition.

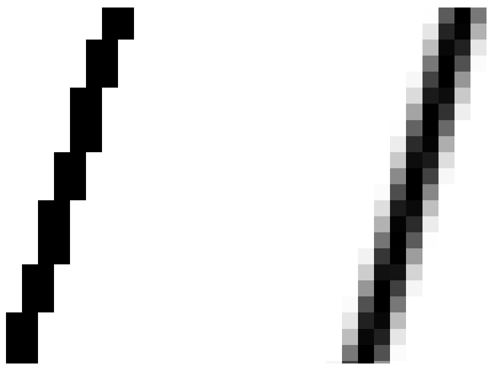


Figure 2.2: Image showing a close-up of a line without anti-aliasing (left) and with anti-aliasing (right) [3].

2.1.2 Ray Tracing

A *ray* is a line starting from a point, and going in a direction. Purely mathematical rays have infinite length, but for practical applications the length can be limited to some distance. By tracing a ray from a given point in a given direction and checking what (if anything) it intersects within its length limit, we can use that information for a number of rendering techniques. The most basic case of using ray tracing for determining the color of the pixels goes as follows: A ray is cast from every pixel into the scene. If the ray intersects at least one object in the scene, we calculate the color of the object closest to the ray origin and set that color to be the color of the pixel from which the ray was launched.

We can also fire a second ray from every hit point towards the light source, to determine whether or not the original hit point is in shadow. If the secondary ray (called a *shadow ray*) intersects any geometry between the original hit point and the light source, the original hit point is in shadow. An illustration is shown in **Figure 2.3**.

This approach can be expanded to include transparent materials, participating media, bounce lighting, and much more.

2.1.3 Light Probes

A light probe is essentially a data structure placed at a point in space that stores information about how light that passes through it behaves, so that this information can be used for

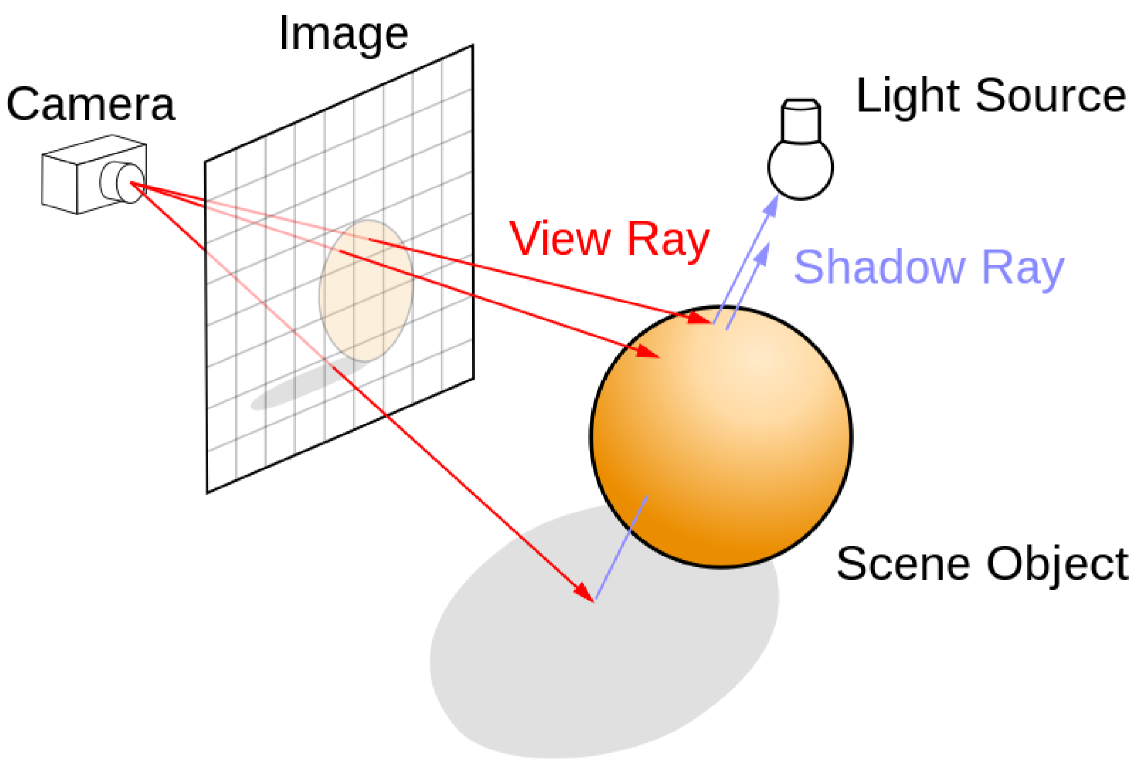


Figure 2.3: Diagram illustrating the fundamentals of ray tracing [8].

lighting. This can be done in a number of different ways, and the specific use case that comes into play for this thesis is explained further on in Section 3.2 (*Probes*).

2.1.4 Surfels

Surface elements, or *surfels*, are used to discretize a surface into a manageable number of smaller units. A surfel is placed on a surface in 3D space, and typically stores its own location, radius, and some information about the surface at the point where it is placed, such as the albedo color and the normal. The way surfels are used in this project is explained in Section 3.3 (*Surfels*).

2.1.5 The DirectX API

In order to communicate with, and give instructions to, the GPU, we use the *DirectX* API developed by Microsoft. This allows us to write code that can be executed on the GPU, to accelerate our rendering applications. Full specifications of the API can be found at Microsoft's website [11].

2.2 Related Work

Ubisoft Massive used a GI system called *Precomputed Radiance Transfer* (PRT) [13] in their game *Tom Clancy's The Division* (2016), which was the inspiration for the system built in this thesis.

The PRT system used light probes to achieve global illumination in large and complex scenes, with a lot of light sources and changing time of day. This approach was very successful, and yielded good visual results while keeping costs low enough for real-time frame rates. However, the big limitation of this system was that it required the main geometry of the scene to be static, so no walls could be broken or moved, and nothing moving could contribute to the GI.

EA SEED presented *Global Illumination Based on Surfels* (GIBS) [2] at SIGGRAPH 2021, which is a dynamic surfel-based GI system. GIBS discretizes the surfaces of a scene in screen space on the fly, scaling the surfels so that all geometry is sufficiently covered from the perspective of the camera. Lighting calculations are then cached in the surfels and amortized over time. All surfels are of roughly the same size in screen space, meaning that there are more surfels per unit of area closer to the camera than far away from it. As the player moves to new areas of the scene, new surfels are spawned to cover the new geometry, and surfels that are not in use are eventually discarded. GIBS is now a part of the *Frostbite Engine* [6].

Chapter 3
Approach

This chapter gives a high-level overview of how the DDGI system implemented for this thesis works, without going into the details of the implementation.

3.1 Coordinate System

We used a right-handed orthogonal coordinate system for the entirety of the project. For brevity, we will be referring to the directions along the basis vectors of the coordinate system as the *cardinal directions* going forward.

3.2 Probes

Probes are placed in a regular 3D grid around the scene to be rendered. Intelligent placement of probes is a whole research area in itself [4], which we will not be exploring here. Instead, the geometry is hand-placed in a such a way that it is reasonably well covered by the probes. **Figures 3.1 - 3.2** show how the probes are placed in the two scenes used for evaluation of the system in Section 5 (*Evaluation*). Each white marker shows the location of a probe.

Each probe stores the colors *coming from* the eight directions `(1, 1, 0)`, `(0, 1, 1)`, `(-1, 1, 0)`, `(0, 1, -1)`, `(1, -1, 0)`, `(0, -1, 1)`, `(-1, -1, 0)`, and `(0, -1, -1)`. We will be referring to these eight directions as the *principal directions*. This octagonal representation was chosen because it is common for game scenes to have a lot of vertical and horizontal surfaces, such as walls and the ground. Having the probes store diagonal directions ensures that light bouncing on vertical surfaces will be able to reach the surrounding horizontal surfaces, and vice versa. In other words, this configuration of principal directions is intended to give good results in common scenes. **Figure 3.3** shows an illustration of a probe, with the principal directions marked.

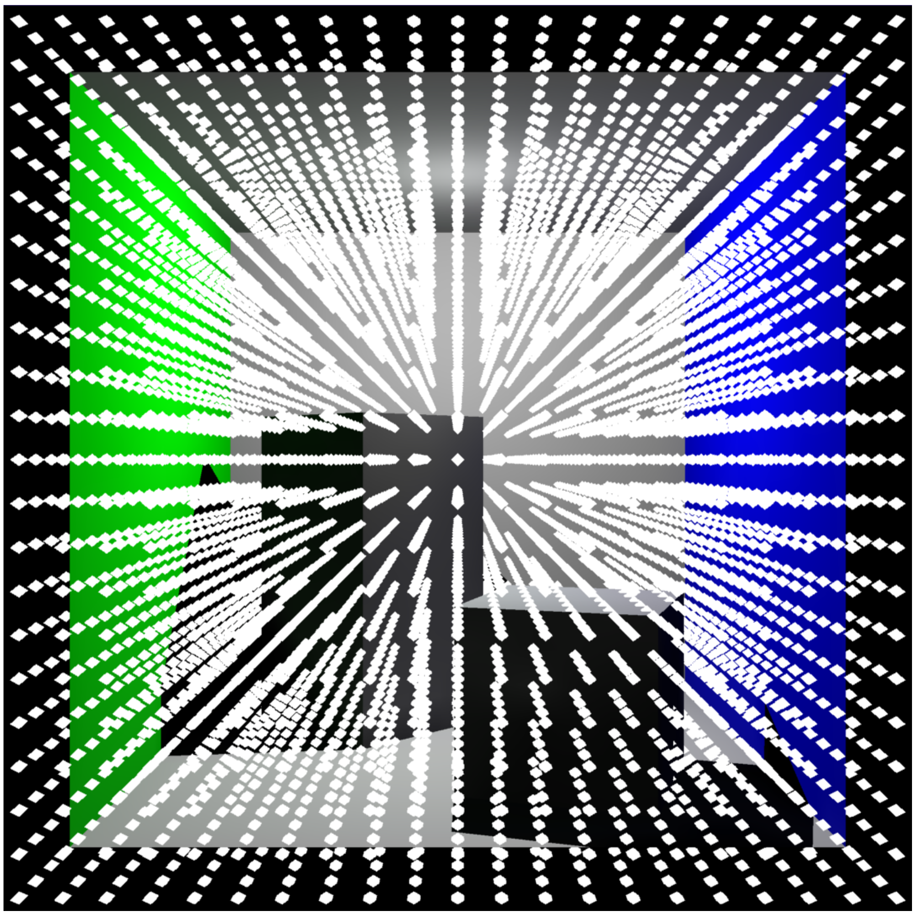


Figure 3.1: Image showing the placement of probes in the *Cornell Box* scene. Each white marker shows the location of a probe.

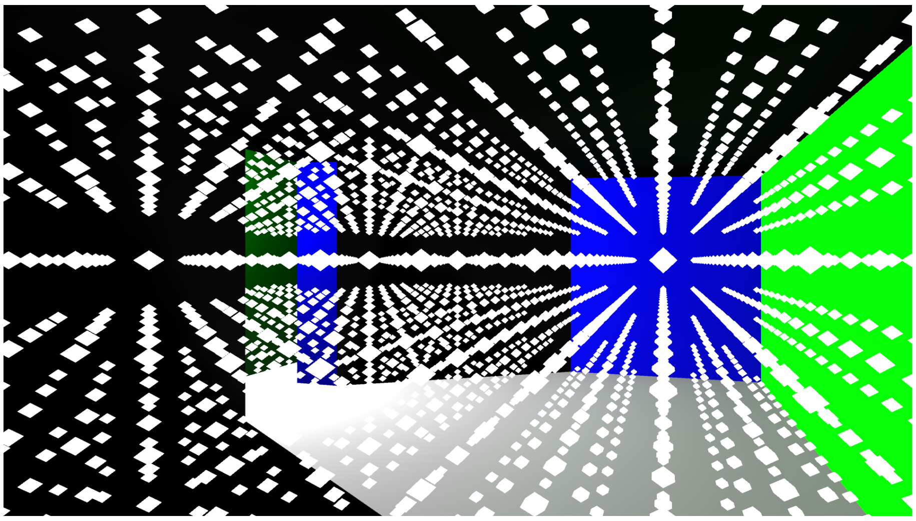


Figure 3.2: Image showing the placement of probes in the *Room With Door* scene. Each white marker shows the location of a probe.

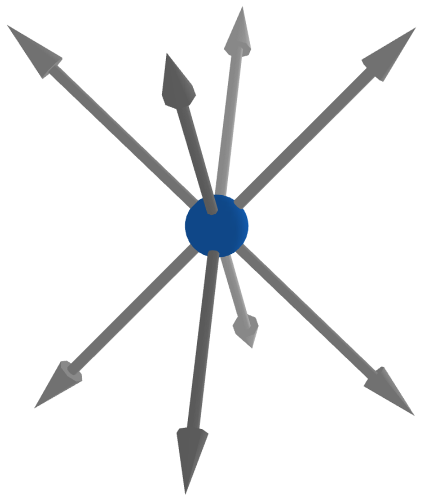


Figure 3.3: Illustration of a probe, with the eight principal direction vectors shown as arrows.

When lighting a pixel, the primary hit shader looks up the 8-neighborhood of probes around the hit point, and uses the colors stored in those probes to calculate the second-order illumination contribution. This is added to the primary lit color of the pixel, as calculated according to Section 4.1.3 (*Shading*).

3.3 Surfels

To calculate what colors to store in a probe, we use surfels. Each surfel consists of a color, a position in space, and a normal, and each surfel belongs to exactly one probe. We do not store a radius for the surfels, but instead use the distance from the probe they belong to to scale their contributions to the lighting. The surfels are placed from the probes using ray tracing: Every probe casts a ray for each surfel it needs to place, in a pattern that roughly covers a sphere around the probe. A simplified illustration of the surfel placement can be seen in **Figure 3.4**.

The pattern is hand-picked, and depends on the number of surfels that are to be placed. The length of the rays is limited such that surfels are placed at most one layer of the probe grid away from the probe they belong to. This is done to make the re-placing of surfels faster, in ways that are explained Section 3.6 (*Caching and Updating Surfels*) below.

For every ray, we check if there was a geometry hit. If there was, we calculate the albedo color, the world space position, and the normal of the geometry at the hit point, and store these data in the surfel. If there was no hit, the surfel is marked as a miss. For every probe, each surfel belonging to it is also considered to belong to exactly one of the probe's principal directions, namely the principal direction which it is the closest to. The hand-picked pattern used for placing the surfels is constructed such that, for each of the principal directions, 1/8 of the surfels belong to it.

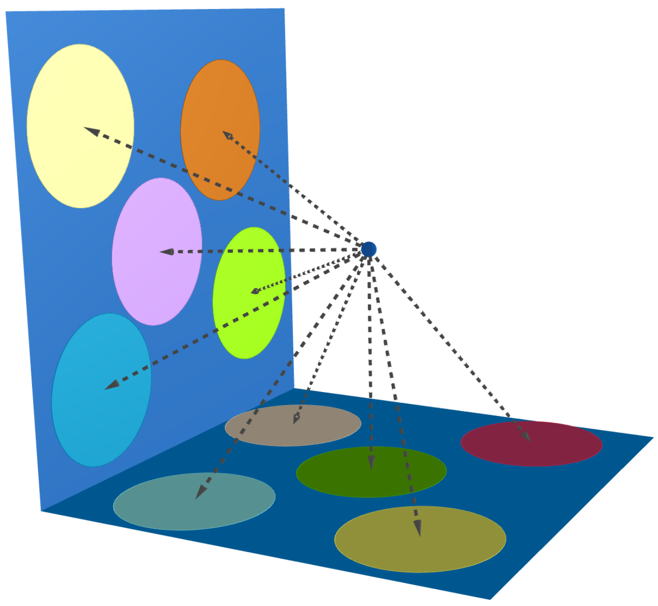


Figure 3.4: Illustration showing a probe, represented as a blue sphere, placing surfels, represented as colored circles, by casting rays, represented as dashed arrows. This is not any of the placement patterns used in the project, and the colors of the surfels are just to make them stand out from the background - in practice they would all be storing the blue color of the surface they are placed on.

3.4 Lighting The Probes

When the surfels have been placed they can be used to light the probes, i.e. calculate what colors to save for each of the principal directions. This is done in the following manner: For each of the principal directions, we calculate the lit diffuse color of the valid surfels that belong to it. We then treat these surfels as lights, and the principal direction vector as a surface normal, and calculate the lighting contributions of the surfels on the probe. Here we also divide the lighting contribution by the squared distance from the surfel to the probe, to account for light falloff over distance according to the inverse square law [1]. We also divide the contribution from each surfel by the total number of surfels belonging to that principal direction. Surfels marked as misses are treated separately, as explained in Section 3.5 below.

The probes are re-lit every frame, to account for changes in lighting conditions (position and color of light, objects casting shadows, etc.). This means that we can move the light source around however we like, and have the GI keep up, without having to do any new ray tracing at all! This is very good for performance in static scenes, where the geometry does not move.

3.5 Color Propagation

When lighting the probes, some (in practice most) of the surfels will be marked as misses, when the ray attempting to place the surfel did not hit any geometry within its length limit.

For these surfels we do not calculate any diffuse lighting, but we instead look to the neighboring probe in the principal direction, and use the color that *it* has stored as coming from the same principal direction. **Figure 3.5** shows which neighbors a probe will find along the principal directions, and **Figure 3.6** illustrates the color propagation in two dimensions and along only one of the principal directions.

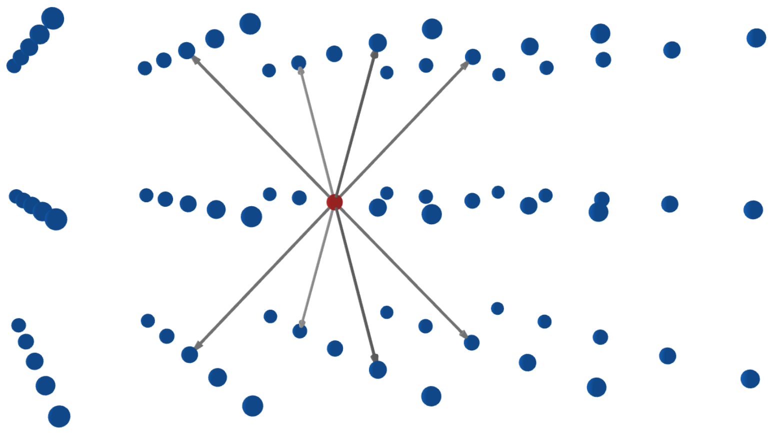


Figure 3.5: Illustration showing which neighboring probes a given probe, marked in red, will find along the principal directions, marked as grey arrows.

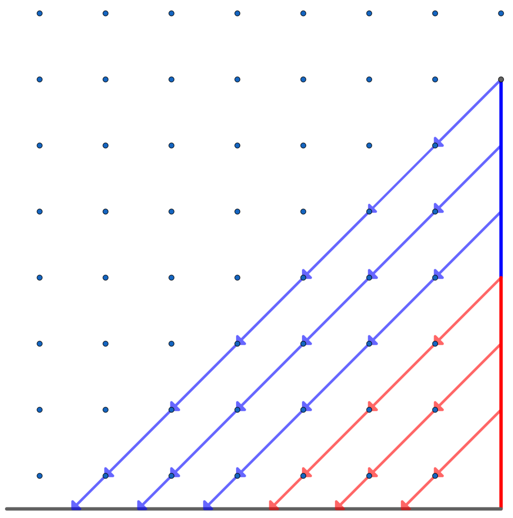


Figure 3.6: Illustration of light propagation through the probe grid, simplified to two dimensions and one principal direction. The light bounced from the blue and red wall to the right is propagated through the probes, represented by the blue dots.

This color is also divided by the square of the distance between the two probes, to account for the light falloff over distance. We then use the surfel as a light in the manner explained above, but with the neighboring probe's color instead.

This is an essential part of the system, since it is what allows bounced light to propagate through space further than one layer of the probe grid.

3.6 Caching and Updating Surfels

When the surfels are placed, they are stored in a buffer so that they can be read from there instead of being re-placed every frame, as long as no geometry has changed. This saves on the costly ray tracing operations and improves performance by a great deal, as will become evident in Chapter 5 (*Evaluation*). In order to make the system dynamic and accommodating of changing geometry, we need a way to detect if geometry has changed, so that we can update the surfels that have been affected. This is done as follows: For every dynamic object in the scene, we keep track of the (smallest) AABB that covers it. For every frame and every object, the application compares the AABB of the previous frame to the AABB of the current frame (after any animation). If the AABBs are different, that means the object has moved, and the application tells all the probes that are placed inside either of the AABBs to re-place their surfels. A bias is also added when checking the AABBs, so that probes that are placed outside the AABBs with a margin of less than the distance between two layers of the probe grid also re-place their surfels. This is to make sure that we update the surfels of all probes that could have placed surfels on the object before it moved, and all probes that can place surfels on the object after it has moved.

This technique ensures that we have a very low cost for the GI when the geometry is static, since we only have to re-light the probes from the surfels already placed, and that we only have to re-place a subset of the surfels when something does change. Since the bounced light is propagated across the scene, changes in geometry can have effects on the scene as a whole, without having to re-place the surfels of all probes. This is the key to the efficiency of the system.

Chapter 4
Implementation

The DDGI system described in this thesis was implemented in DirectX 12 in the C++ and HLSL languages on a Windows 10 PC at the Department of Computer Science at LTH. This chapter goes into more detail about the technical implementation, but does not delve into the actual code. The interested reader is referred to the full source code, available in Section 1.4 (*Source Code*).

4.1 The Core Renderer

The foundation on which the DDGI system is built is a very simple ray tracing renderer, which in turn is based on a sample from Microsoft [10] and a tutorial by Nvidia [9]. This core renderer does all the essential so called "boiler plate" operations of setting up the GPU for real time rendering, loading vertices and index lists for the geometry, compiling shaders, sending data to buffers, dispatching rays and compute shader instances, etc. This is not very interesting, while still being very technically complex, so we will avoid these tedious parts as much as possible. However, understanding of some of the basic components is required to understand the interesting bits, and so an overview is provided below.

4.1.1 Scene Construction

The scenes created for this project all consist of a square base plane of 400 by 400 length units, with all additional geometry added above it. The base plane of the scene is always centered around the origin. All dynamic geometry is represented by instances of a base cube, which is transformed and reshaped by way of multiplication with a transformation matrix. The colors of objects' surfaces are defined by the colors stored in their vertices.

All scenes contain a single point light source, represented as a position and a color, that can be moved around.

4.1.2 Basic Rendering loop

The rendering loop begins with the invocation of the *RayGen* shader, which dispatches the *primary rays*, one per pixel. The rays are cast straight into the 3D scene, from positions representing the pixel positions on screen, to answer that fundamental question: "What color is this pixel?" If no geometry is hit, the *Miss* shader is invoked, which always returns the color black. If a ray hits a piece of geometry, the *Hit* shader is invoked instead.

4.1.3 Shading

The Hit shader calculates the diffuse color at the ray hit point, using the geometry and color information specified by the vertices, and returns this as the color of the pixel corresponding to the ray from which the Hit shader was invoked. The diffuse color L_{out} is calculated as

$$L_{out} = L_{light} \cdot L_{albedo} \cdot max(n \cdot l, 0) \cdot S + L_{ambient},$$

where L_{albedo} is the albedo color of the geometry, n is the surface normal of the geometry at the hit point, l is the vector from the hit point to the light source, L_{light} is the color of the light from the light source, S is the *shadow term*, and $L_{ambient}$ is the *ambient term*.

The shadow term is calculated by casting an additional ray from the primary ray hit point towards the light source, limited in length to the distance between the two. If the shadow ray hits some geometry, we know that the primary ray hit point is in shadow. If the shadow ray doesn't hit anything, we know the the primary ray hit point is in direct light. If it is in shadow, its color is darkened. For more realistic soft shadows, a small random offset can be added to the direction of the shadow ray to represent the physical size of the light source. We can then cast more than one shadow ray per primary ray hit, all with the random offset in different directions, and average them together. This is a lot more computationally expensive, but yields very good looking results. More advanced modern video games have implemented techniques to amortize the cost of this soft shadow method over time, which, combined with modern de-noising, can result in real-time frame rates with very impressive looking shadows [14]. This is not something that will be explored in this thesis, however.

The ambient term is simply a very small value added to all three channels of the lighting of all primary ray hits, to slightly bring out the bits of geometry that did not receive any direct or bounced light at all. This is not strictly correct, but improves the result slightly at essentially no extra cost.

4.1.4 Supersampling Anti-Aliasing

SSAA is achieved by dispatching the *RayGen* shader four times per pixel, each with a different slight offset from the pixel center. This will give four slightly different color values, that are then averaged together to provide anti-aliasing to the final image. The cost of the primary ray trace is increased, since we run the *RayGen* shader four times as many times per frame, but for our purposes the increase in visual quality is worth this trade-off.

4.1.5 Animation

The transformation matrix applied to every object in a scene can be updated for every frame, either manually by the user or according to some function of time, enabling animation.

4.2 Probe Update Shader

The DDGI system is run from a compute shader, which is invoked once per probe per frame. We will call this shader the *probe update shader*. The probe update shader is responsible for updating and relighting the probes, placing the surfels, and propagating light between probes. The lighting of pixels is still done in the primary ray Hit shader, which only reads from the probes but does not write to them.

4.3 Probe Implementation

This section describes how the probes are implemented.

4.3.1 Structure

The probes are represented as a struct (in both the C++ and HLSL code) containing an array of eight colors. The world space position of a probe is decided by its location in the probe buffer, as explained below.

4.3.2 Placement

The probes are laid out in m layers in the y-direction, each layer being n by n probes in the xz-plane. The probes are placed in a regular grid, i.e. the distance between two neighboring probes along any of the cardinal axes is always the same. There can be any (real, positive) number m of layers in the y-direction. The distance between the layers in the y-direction is the same as the distance between the probes along the cardinal axes within the layers.

The bottom probe layer is placed symmetrically around the origin, with all additional layers perfectly above it (in the positive y-direction). That means that probe positions can have negative x and z values, but only non-negative y-values.

We will consider any points in the 3D space that are outside of the volume covered by the probe grid to be invalid, and the behavior of the system at such points is undefined.

4.3.3 Access

It is very important to be able to access the eight probes surrounding a point in space quickly, which means that e.g. searching through the positions of all probes to find the closest ones is ruled out. Instead, we store the probes layer by layer in a one-dimensional array, with each row in the z-direction being contiguous. For an n by m by n probe grid, the first n probes in the array are the first line in the z-direction of the first layer in the y-direction. The next n probes in the array are the second line of the first layer, etc. until we reach $n \cdot n$ probes. The next n probes in the array are then the first line of the second layer, the next n are the second line of the second layer, etc. This then goes on for m layers. **Figure 4.1** shows an example of an 11 by 4 by 11 probe grid, marking the very first element in cyan, the rest of the first line in yellow, and the rest of the first layer in green. **Figure 4.2** shows the memory layout of this grid as stored in an array, marking the probes with the same colors.

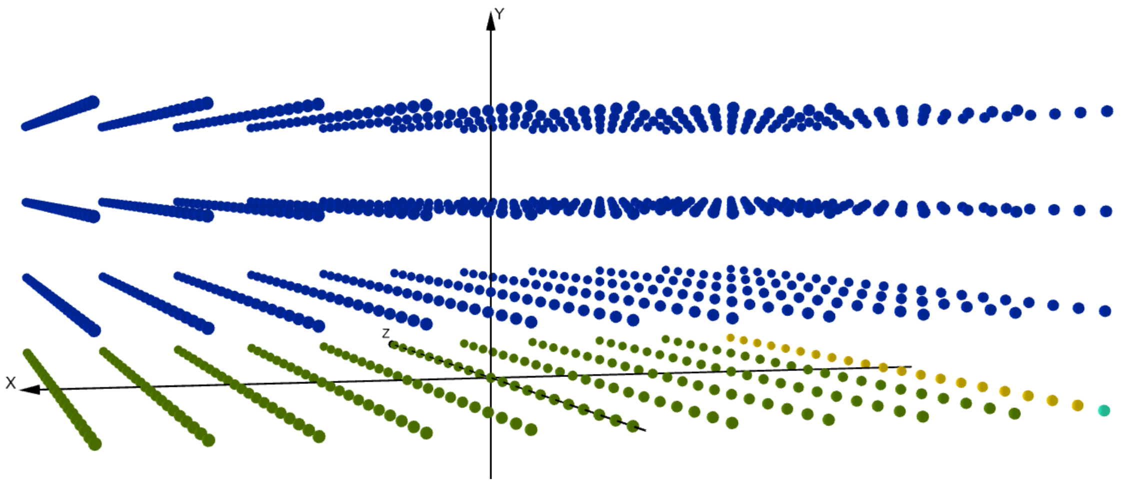


Figure 4.1: Illustration of the probe grid layout. The very first element is marked in cyan, the rest of the first row in the z-direction is marked in yellow, and the rest of the first layer is marked in green. The probes are evenly spaced out, as described in Section 4.3.2 (*Placement*) above.

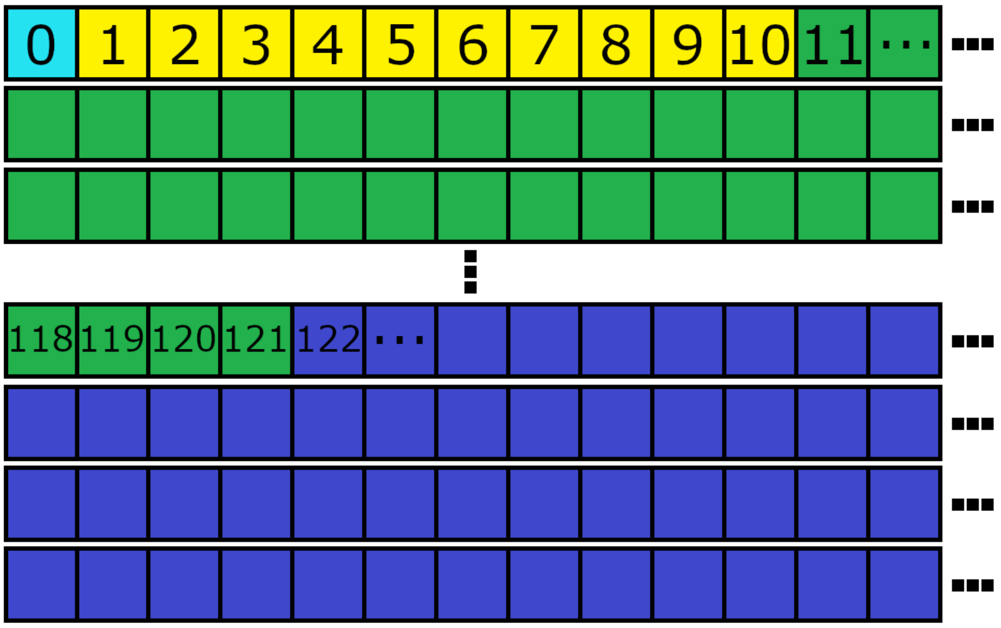


Figure 4.2: Illustration of the probe memory layout for the same probe grid as in Figure 4.1. The image shows a single one-dimensional array, broken up on multiple lines for readability.

```
 1    // Move and clamp coordinates for mapping to probe buffer.
 2    float x = clamp(hitWorldPos.x + basePlaneSideLength / 2, 0,
         basePlaneSideLength);
 3    float z = clamp(hitWorldPos.z + basePlaneSideLength / 2, 0,
         basePlaneSideLength);
 4    float y = clamp(hitWorldPos.y, 0, nbProbeLayers * stepSize);
 5
 6    float xSnappedDown = x - (x % stepSize); //< x rounded down to
         nearest stepSize.
 7    float zSnappedDown = z - (z % stepSize); //< z rounded down to
         nearest stepSize.
 8    float ySnappedDown = y - (y % stepSize); //< y rounded down to
         nearest stepSize.
 9
10    int xIndexDown = (int)(((xSnappedDown) / stepSize) * sqrt(
         nbProbesPerLayer));
11    int xIndexUp = (int)(((xSnappedDown) / stepSize + 1) * sqrt(
         nbProbesPerLayer));
12    int zIndexDown = (int)((zSnappedDown) / stepSize);
13    int zIndexUp = (int)((zSnappedDown) / stepSize + 1);
14    int yIndexDown = (int)(((ySnappedDown) / stepSize) *
         nbProbesPerLayer);
15    int yIndexUp = (int)(((ySnappedDown) / stepSize + 1) *
         nbProbesPerLayer);
```

Listing 4.1: Code snipped showing the process for finding the 8-neighborhood of probes around a point in space.

This scheme makes finding the probes surrounding a point in 3D space very fast. **Listing 3.1** shows the HLSL code used for going from a point in world space (e.g. a primary ray hit point) to the 8-neighborhood of probes around it.

Since negative values are allowed in the x and z directions, but array indices can only be non-negative, we need to move all values in such a way that they are guaranteed to be non-negative for all valid points in space. Lines 2-4 add half the length of the side of the base plane to the x and z values, ensuring that all valid points are certain to have non-negative x and z values. We also clamp the values to be between zero and the length of the side of the base plane. Lines 6-8 round the coordinates down to the nearest `stepSize`, which is the distance between any two probes along any of the cardinal directions. The final lines of the listing calculate the indices of the probe one step "below" and the probe one step "above" the point for each of the three cardinal directions.

4.4 Surfel Implementation

Surfels are used to cache the information needed to relight the probes each frame. When the geometry that a surfel is placed on changes, the surfel need to be replaced.

4.4.1 Structure

The surfels are represented as a struct (in both the C++ and HLSL code) containing a color, a position, and a normal, each represented as a floating point array of length three.

4.4.2 Placement

The surfels are placed by casting rays from the probes, as explained in Section 3.3 (*Surfels*). The rays are limited in length such that their longest component along any of the cardinal directions is equal to the distance between any two probes along any of the cardinal directions. This ensures that every probe can cover the whole space around it, up until the next probe layer over, in all directions.

If a ray attempting to place a surfel does not hit any geometry within its length limit, the surfel is marked as a miss by setting its normal to the zero vector (an invalid value). When lighting the probes, we check the normal of each surfel to see which ones are misses and which ones are valid. If a surfel is valid it is lit as normal, but if it is invalid we instead use the light propagated from the neighboring probe in the principal direction which the invalid surfel belonged to, as explained in Section 3.5 (*Color Propagation*).

4.4.3 Lighting Surfels

The surfels are lit using the same method as with the primary rays' Hit shader, described in 4.1.3 (*Shading*). Their contribution to the probe's lighting is also divided by d^2, where d is the distance between the surfel and the probe, to account for light falloff according to the inverse-square law [1].

4.5 Axis-Aligned Bounding-Boxes

AABBs are used to detect changes in the geometry.

4.5.1 Structure

The AABBs are represented as a struct containing an array of 6 floating point values. These represent the limits of the bounding box in the positive x, negative x, positive y, negative y, positive z, and negative z directions respectively. An AABB can be constructed such that it contains no points, by setting the positive limit for each coordinate to be less than the negative limit.

4.5.2 Construction

To construct the AABB around an object, we simply set the limits for every coordinate of the AABB to be the greatest and smallest value for that coordinate of any vertex in the object. The upper limit in the x-direction is set to be the greatest x-value of any of the vertices of the object, the lower limit in the x-direction is set to be the lowest x-value of any of the vertices of the object, etc.

4.5.3 Comparisons

To find if a point is within an AABB, we simply check if the coordinates of the point are within their respective limits in the AABB.

4.6 Detecting Changes to the Geometry

The AABBs are used to detect if geometry has moved, and to tell the affected probes that they need to update their surfels since they might now be invalid. For every object in the scene that can move (which can be all of them), the application keeps track of the current AABB after all animation has been executed, and the AABB for the previous frame. These are compared every frame to see if they differ from each other. If they do differ, both of them are sent to the probe update shader. If they are the same, two invalid AABBs are sent instead, constructed such that they contain no points.

Each probe checks the buffer containing the AABBs every frame, to see if it is contained within any one of them. Probes can place their surfels up to one layer of the probe grid away from their own position, as explained in Section 4.4.2 (*Placement*) above, and we therefore extend the size of the AABBs so that they cover one extra layer of the probe grid in each of the cardinal directions (both positive and negative). This is to account for probes at the edge of the original AABB that might have placed their surfels inside of the AABB, and therefore need updating even though they are outside it.

Chapter 5

Evaluation

This chapter delves into the visual results and execution times of the DDGI system. The execution times were measured on a PC with the specifications shown in **Table 5.1**, at a resolution of 1920 × 1080 pixels.

We could not observe any noteworthy change in the appearance of the DDGI as we increased the number of surfels per probe. This is likely due to the fact that the renderer does not support textured surfaces, so the increased resolution of the probes' lighting did not contribute to the image quality. For this reason, and in the interest of saving space, we only include screen captures of the scenes with 64 surfels per probe.

Component Type	Component Name and Specifications	Manufacturer
GPU	GeForce RTX 2070 SUPER, 8 GB VRAM	Gigabyte/Nvidia
CPU	Ryzen 5 5600X, 3.7 GHz, 35 MB cache	AMD
RAM	Vengeance DDR4, 16 GB, 3200MHz, CL16	Corsair

Table 5.1: Hardware specifications for the computer used to measure execution times.

5.1 Sample Scenes

The two scenes used for evaluation of the system were the classic *Cornell Box* [5], and a purpose built scene called *Room With Door*. For each scene, a pre-defined animation was executed while timing the system, to show how it behaves when all geometry is static compared to when something is moving. In both scenes there is a single point light, and its color is pure white.

5.1.1 Cornell Box

The *Cornell Box* scene consists of a cubical box, with two of its sides colored green and blue, respectively. Inside the box two blocks are placed: one a cube and the other a taller cuboid. The blocks and the four remaining sides of the box are colored grey. The wall that is closest to the camera is not rendered out (since that would obscure the view of the contents of the box), but it does contribute to the bounced lighting. The scene is shown in **Figure 5.1**, rendered without the DDGI system enabled. This scene contains 9261 probes.

Visual Results

Figure 5.2 shows the scene rendered with the DDGI system enabled. **Figure 5.3** shows only the probe lighting contribution, brightened for ease of inspection.

5.1.2 Room With Door

The *Room With Door* scene (unsurprisingly) consists of a room with a door in the left wall, which can be opened and closed (either by a script or manually by the user). The scene is shown in **Figure 5.4**, rendered without the DDGI system enabled, and with the door open. The light source in this scene is placed outside of the room, with most of the light being blocked by the outer walls. The right wall of the room is colored green, and the back wall is colored blue. This scene contains 5292 probes, and is meant to represent a more typical game scene than the *Cornell Box* scene.

Visual Results

Figure 5.5 shows the scene rendered with the DDGI. **Figure 5.6** shows only the only probe lighting contribution, brightened for ease of inspection.

5.2 High Quality Shadows

The renderer implemented for this thesis supports much higher quality shadows than in the screen captures showed so far, by increasing the number of shadow rays cast from each primary ray hit as described in Section 4.1.3 (*Shading*). However, this comes at a severe impact to performance. Due to time constraints, and the fact that shadow rendering was not the focus of the thesis, we did not implement a real-time solution for these high quality shadows. Such techniques are, however, well-known, and implemented in many production video games, such as Remedy's *Control* [14].

To show what the DDGI system is capable of when integrated into a system with more advanced shadow rendering, **Figure 5.7** shows the *Cornell Box* scene rendered without the DDGI, but with high quality shadows at 1000 shadow rays per pixel, **Figure 5.8** shows the scene rendered with the high quality shadows and the DDGI, and **Figure 5.9** shows only the probe lighting contribution, brightened. Please keep in mind that images were *not* rendered at real-time frame rates.

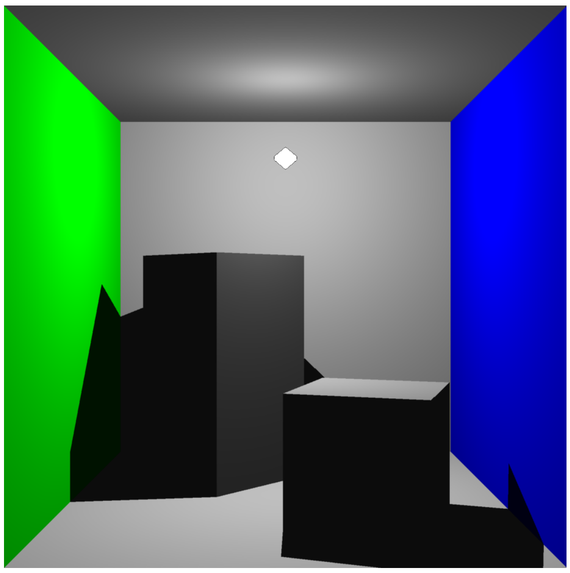


Figure 5.1: The *Cornell Box* scene, rendered without the DDGI. The white marker shows the location of the scene's light source.

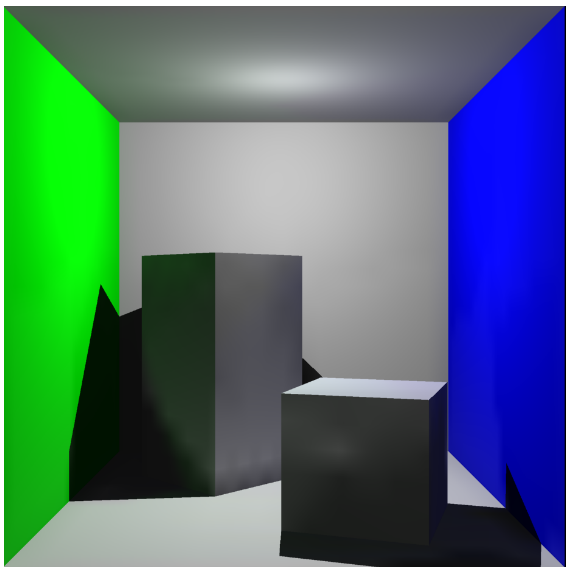


Figure 5.2: The *Cornell Box* scene, rendered with the DDGI. Note the green light spill on the side of the tall block.

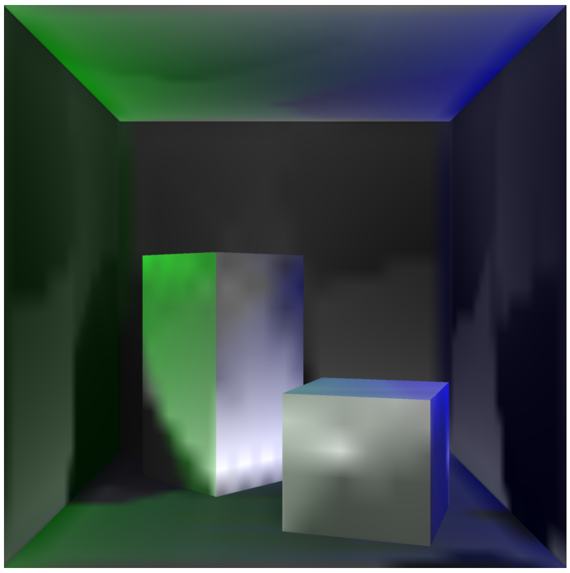


Figure 5.3: The probe lighting contribution in the *Cornell Box* scene, brightened.

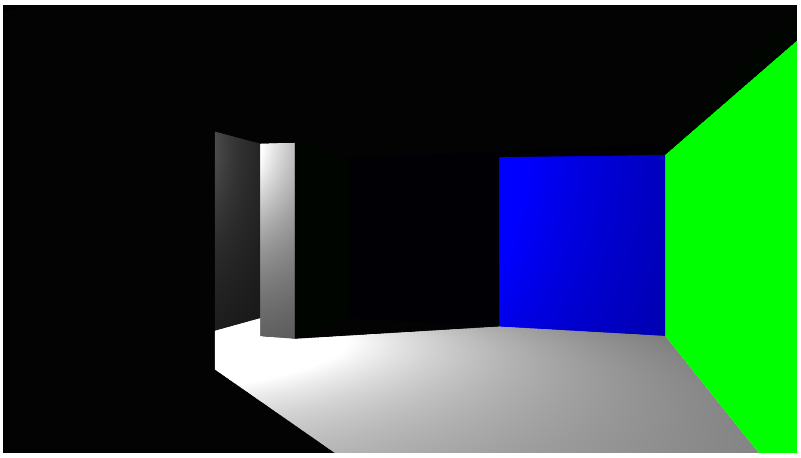


Figure 5.4: The *Room With Door* scene, rendered without the DDGI system.

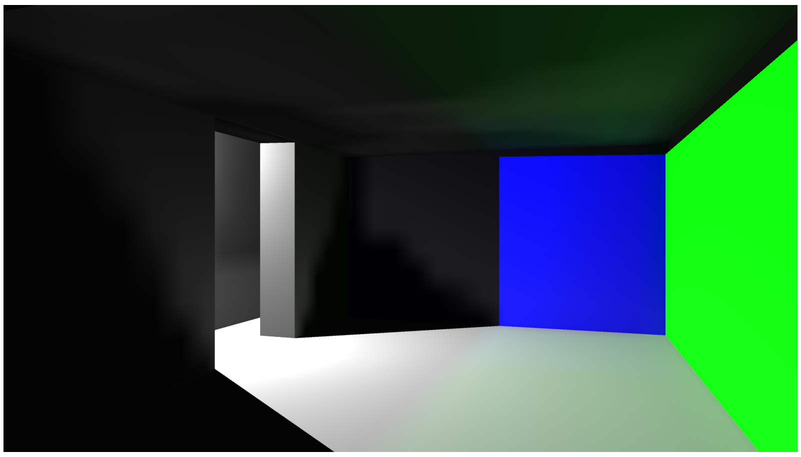


Figure 5.5: The *Room With Door* scene, rendered with the DDGI system.

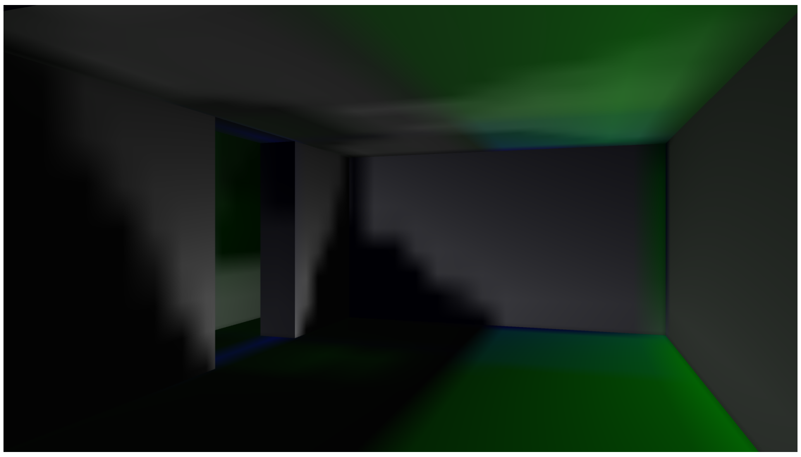


Figure 5.6: The probe lighting contribution in the *Room With Door* scene, brightened.

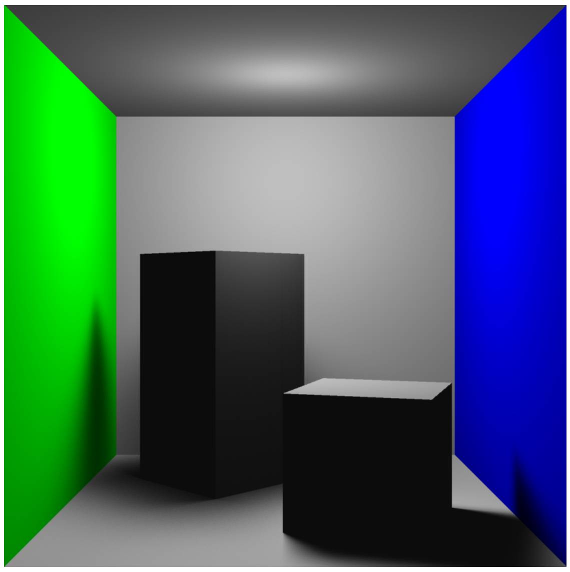


Figure 5.7: The *Cornell Box* scene, rendered with high quality shadows, but without the DDGI.

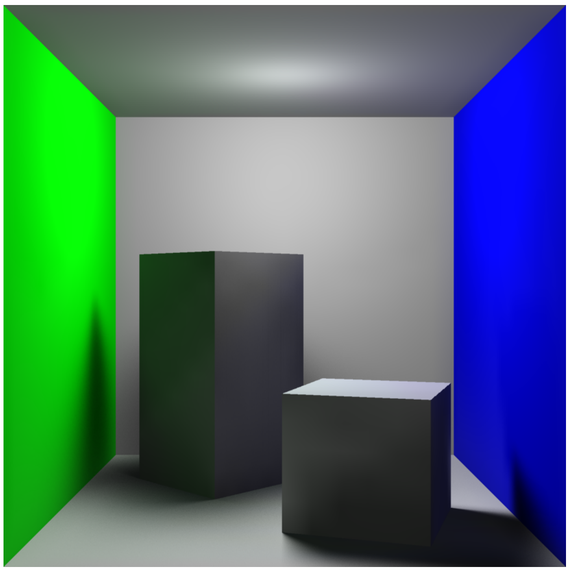


Figure 5.8: The *Cornell Box* scene, rendered with both the DDGI system and high quality shadows.

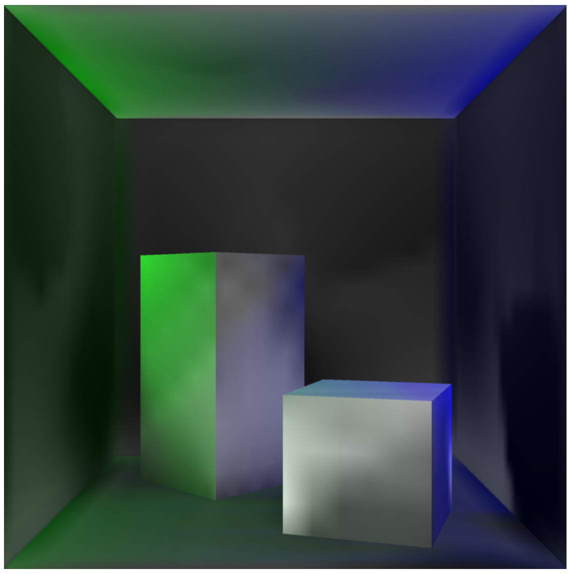


Figure 5.9: The probe lighting contribution in the *Cornell Box* scene, rendered with the high quality shadows and brightened.

5.3 Execution Times

This section looks at the execution times for the probe update shader and the primary ray trace in the different scenes with different configurations of the DDGI system.

5.3.1 Configurations

Six different configurations of the DDGI system were tested per scene, varying the number of surfels per probe, or disabling parts of the system. The configurations are listed in **Table 5.2**. For the *Full GI* configurations, the only parameter that changes is the number of surfels placed per probe. For the *No Surfels* configuration, the caching of surfels is disabled, so all probes are forced to cast 64 rays every frame. *GI Off* simply turns the DDGI system off entirely.

Full GI - 64 surfels per probe
Full GI - 96 Surfels Per Probe
Full GI - 128 Surfels Per Probe
Full GI - 160 Surfels Per Probe
No Surfels - 64 Rays Per Probe
GI Off

Table 5.2: The six different investigated configurations of the DDGI system.

5.3.2 Test Procedure

The scenes were timed for 600 frames each during an automated test sequence. At frame no. 200 an animation began, different for each scene, that lasted for 150 frames until frame no. 350. The scene was then static until frame no. 600, when the application terminated.

We measured the execution time by placing timestamps (using the DirectX query method `ID3D12GraphicsCommandList::EndQuery`) before and after dispatching the probe update shader, and before and after dispatching the primary rays. Comparing the timestamps gave us the elapsed times.

To even out inconsistencies and noise in the execution times, we ran each test case (scene and configuration combination) 100 times, and averaged the execution times for each frame number. This was also done using an automated script.

5.3.3 Execution Time Results

This section presents the results of the execution time measurements for the different scenes.

Cornell Box

The animation for this scene was that the the blocks rotated, completing approximately one quarter of a full revolution. **Figure 5.10** shows the execution times of both the primary ray trace and the probe update shader for the *No Surfels* configuration. **Figures 5.11 - 5.14** show the same measurements for the *Full GI* configurations. **Figure 5.15** shows them for the *GI Off* configuration. **Figure 5.16** shows just the probe update shader execution times for all configurations in the same graph. In these plots we have excluded the first 10 frames, as they tend to be extreme outliers. The dashed vertical lines show the beginning and end of the animation.

Figure 5.17 shows the average execution times for the probe update shader and the primary ray trace when the scene is static versus when it is animated. Here we have excluded the first 80 frames, as these tend to be outliers.

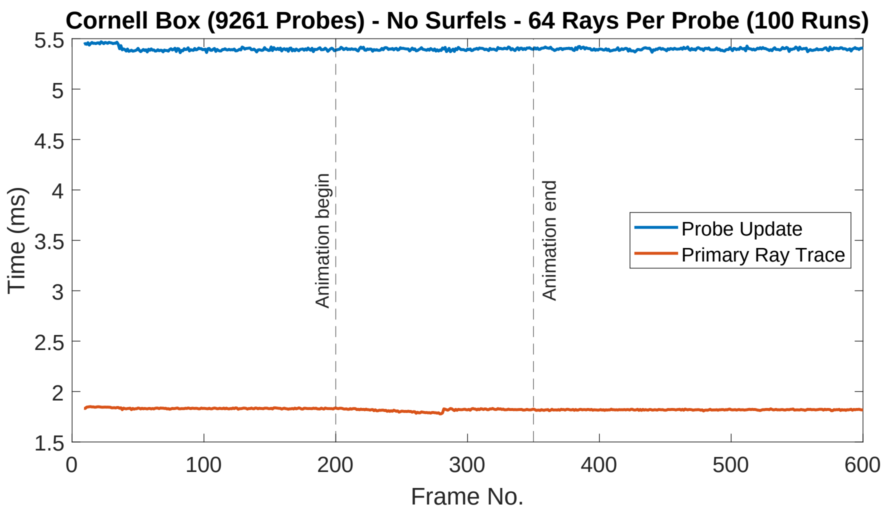


Figure 5.10: Execution times for probe shader and primary ray trace.

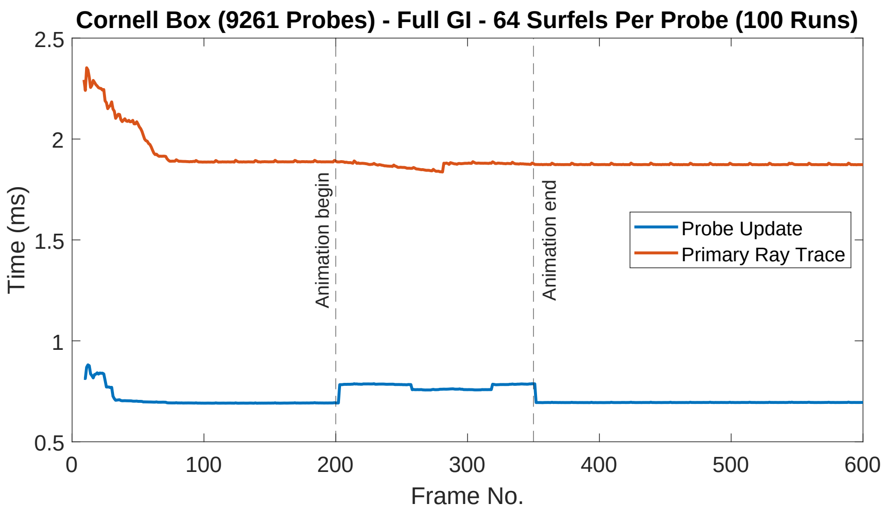


Figure 5.11: Execution times for probe shader and primary ray trace.

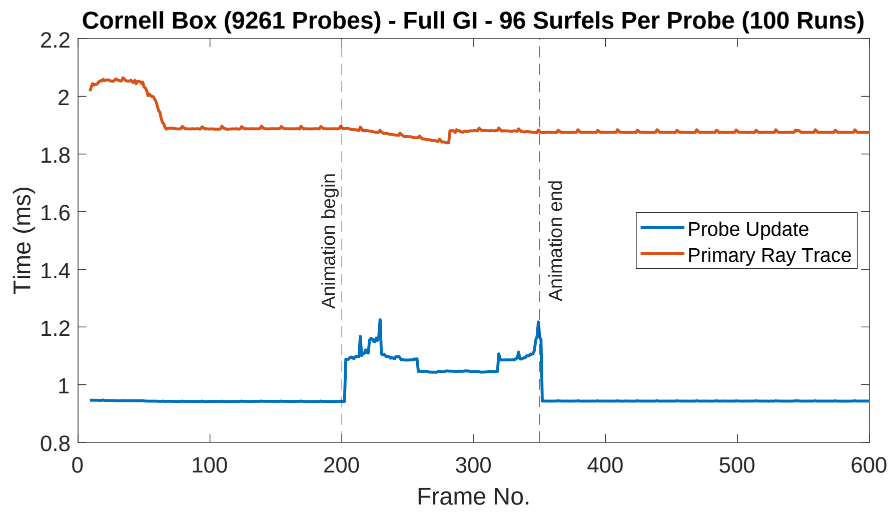


Figure 5.12: Execution times for probe shader and primary ray trace.

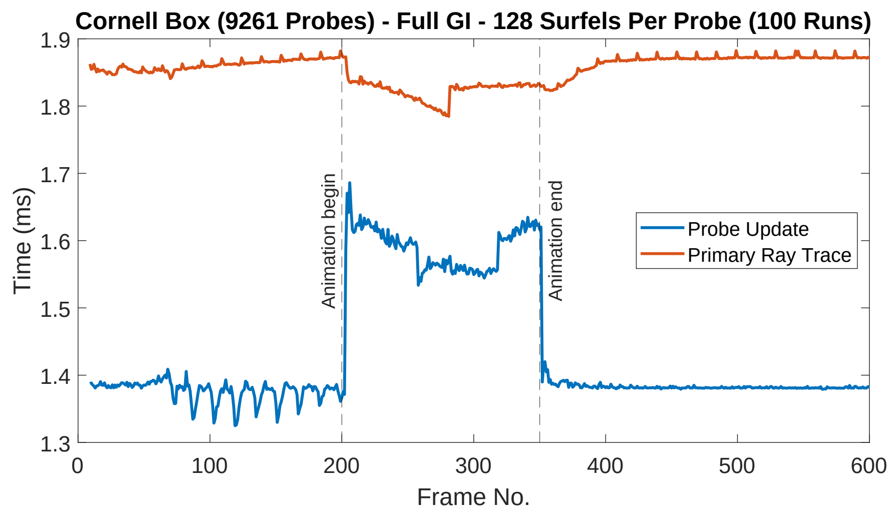


Figure 5.13: Execution times for probe shader and primary ray trace.

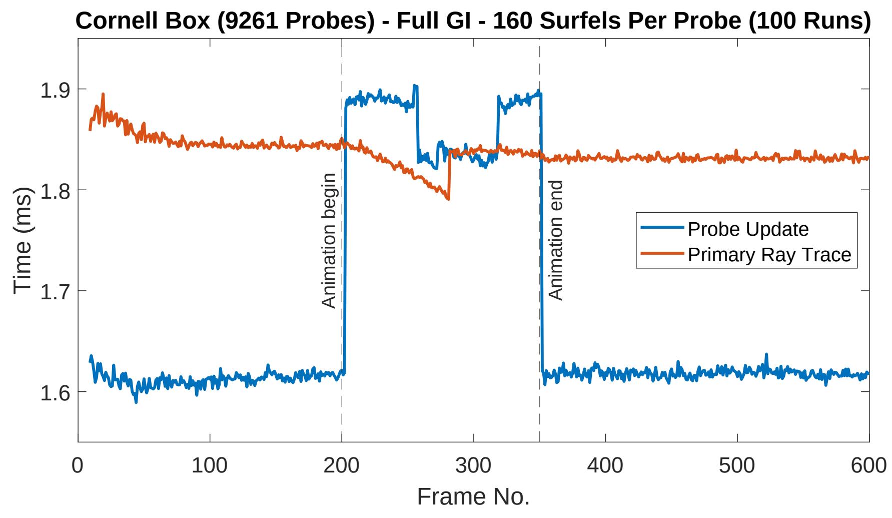


Figure 5.14: Execution times for probe shader and primary ray trace.

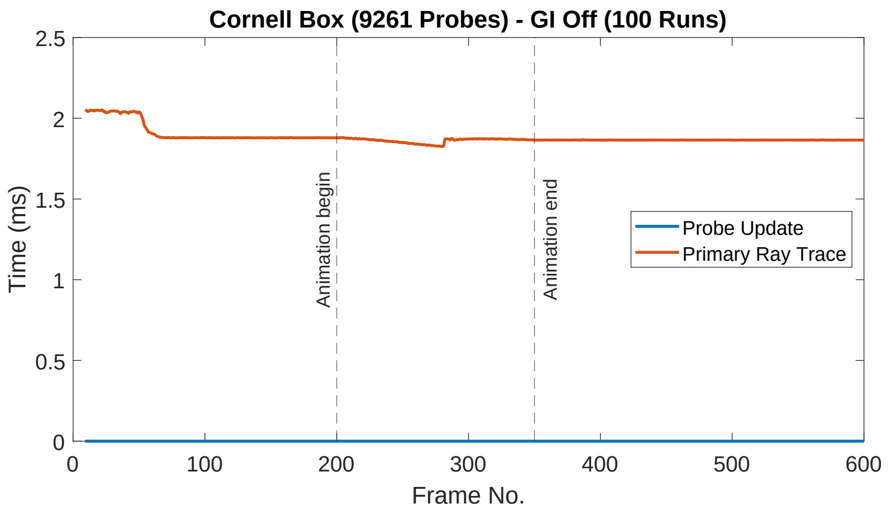


Figure 5.15: Execution times for probe shader and primary ray trace.

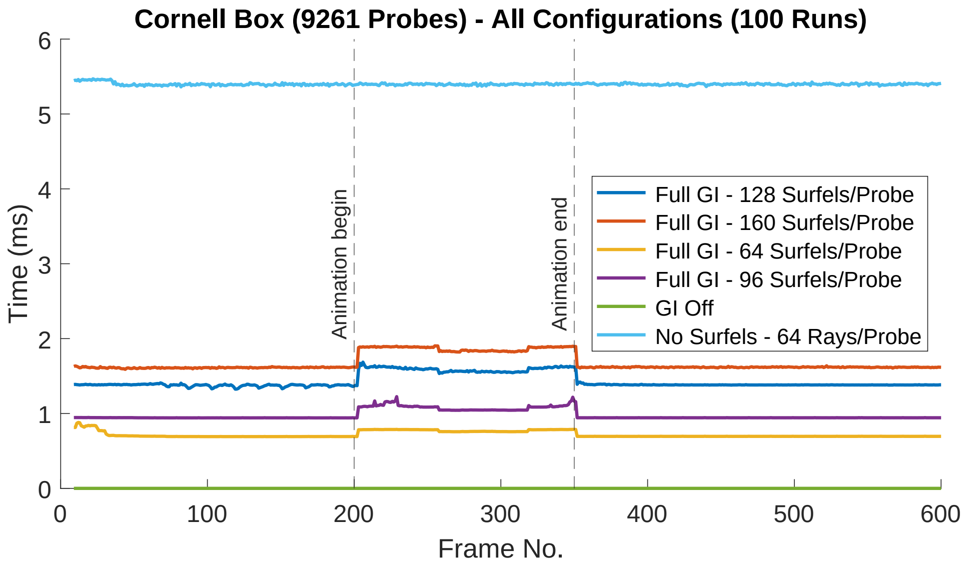


Figure 5.16: Execution times for probe shader.

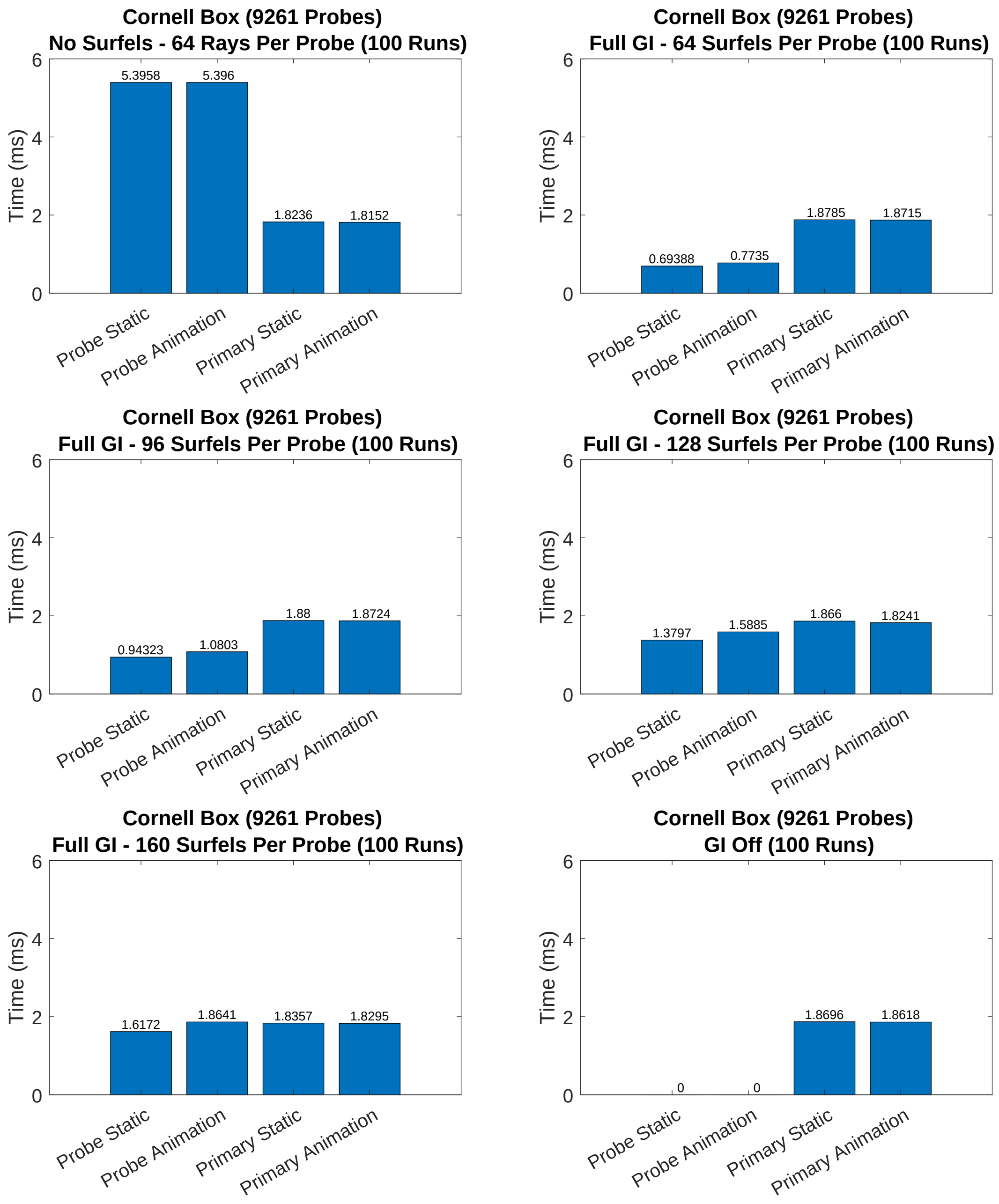


Figure 5.17: Average execution times for the probe update shader and the primary ray trace in the *Cornell Box* scene, for all configurations.

Room With Door

The animation for this scene was that the door to the room slid upwards for 150 frames, from the closed position, to be fully open and let light into the room. **Figure 5.18** shows the execution times of both the primary ray trace and the probe update shader for the *No Surfels* configuration. **Figures 5.19 - 5.22** show the same measurements for the *Full GI* configurations. **Figure 5.23** shows them for the *GI Off* configuration. **Figure 5.24** shows just the probe update shader execution times for all configurations in the same graph. In these plots we have excluded the first 10 frames, as they tend to be extreme outliers. The dashed vertical lines show the beginning and end of the animation.

Figure 5.25 shows the average execution times for the probe update shader and the primary ray trace when the scene is static versus when it is animated. Here we have excluded the first 80 frames, as these tend to be outliers.

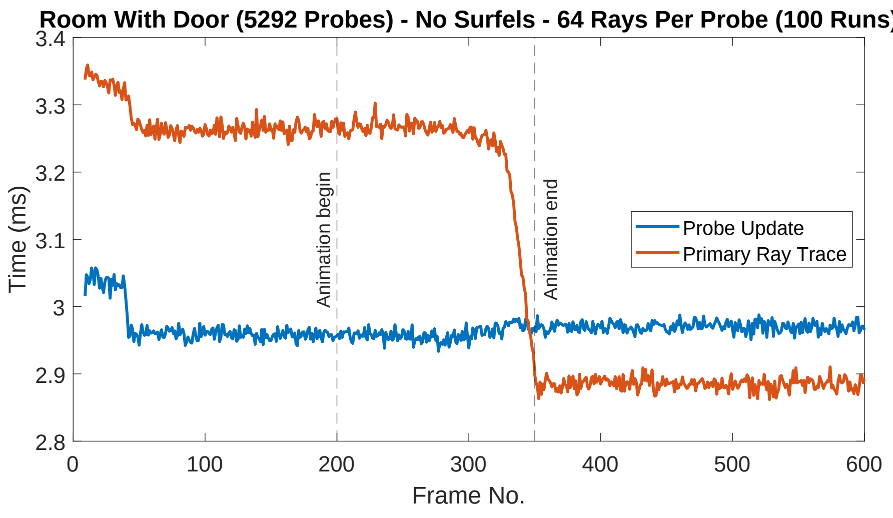


Figure 5.18: Execution times for probe shader and primary ray trace.

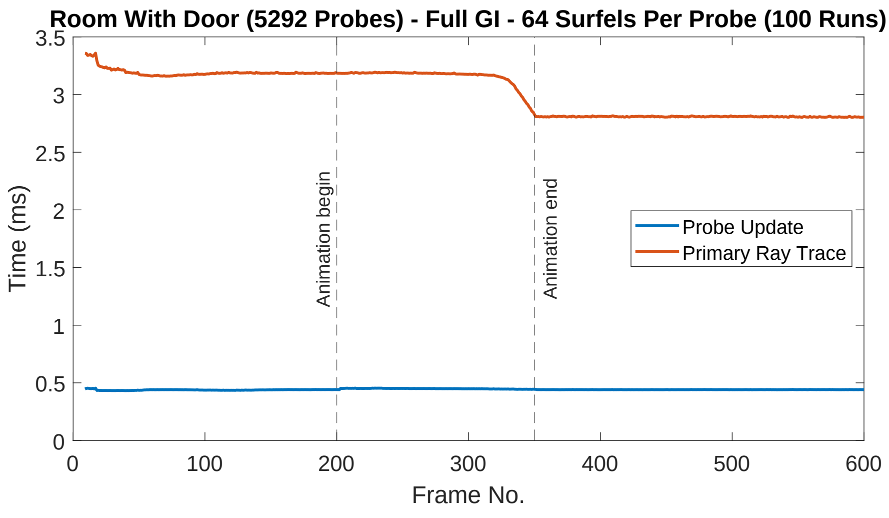


Figure 5.19: Execution times for probe shader and primary ray trace.

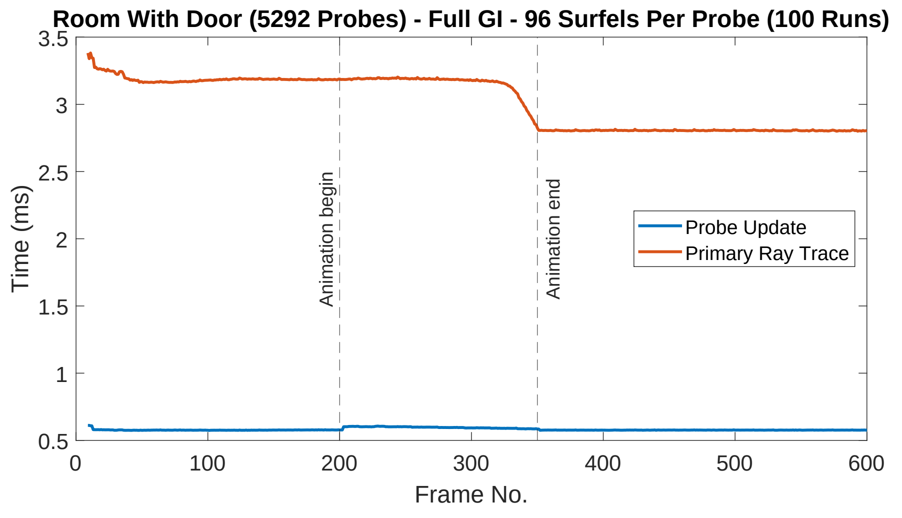


Figure 5.20: Execution times for probe shader and primary ray trace.

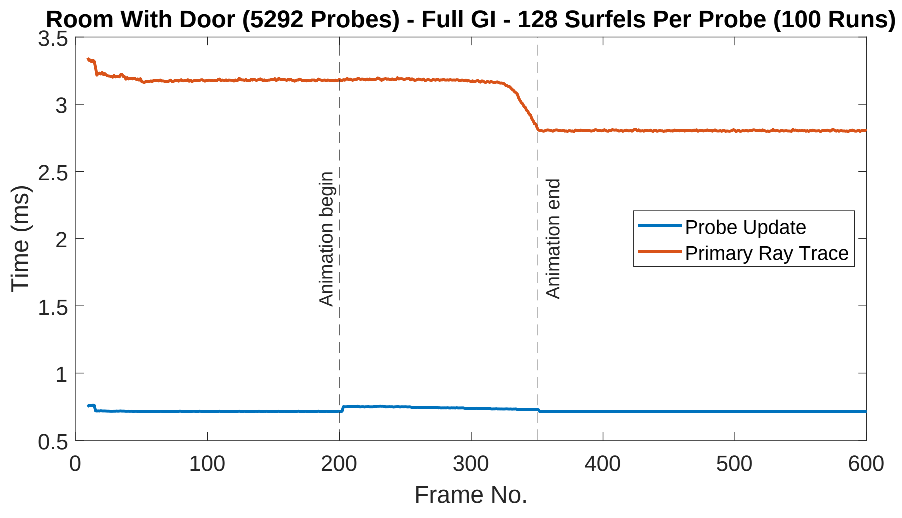


Figure 5.21: Execution times for probe shader and primary ray trace.

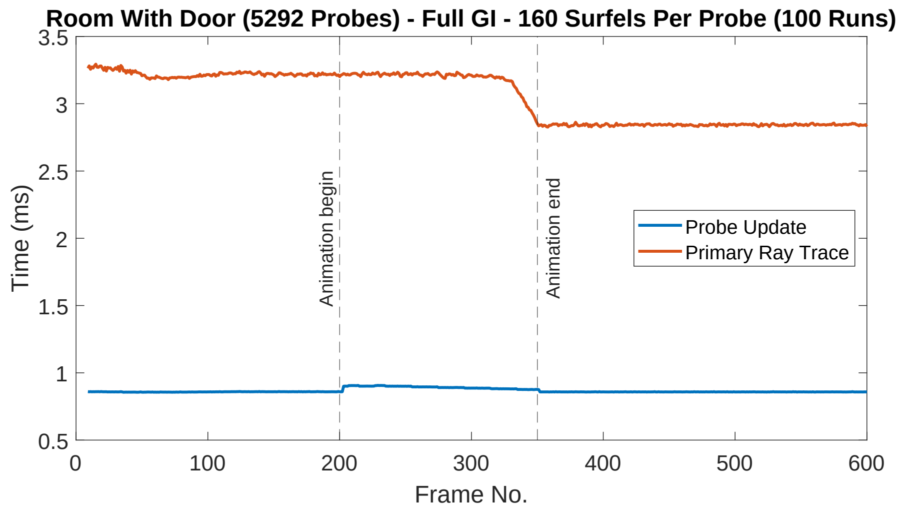


Figure 5.22: Execution times for probe shader and primary ray trace.

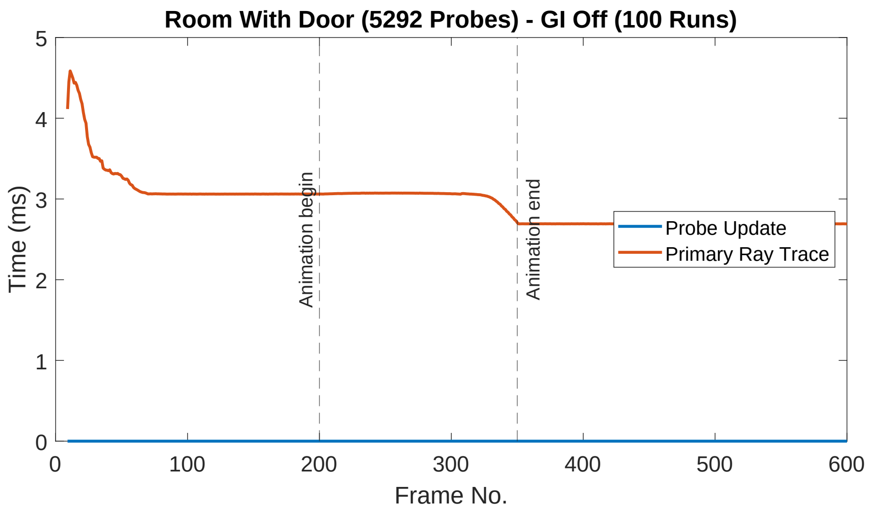


Figure 5.23: Execution times for probe shader and primary ray trace.

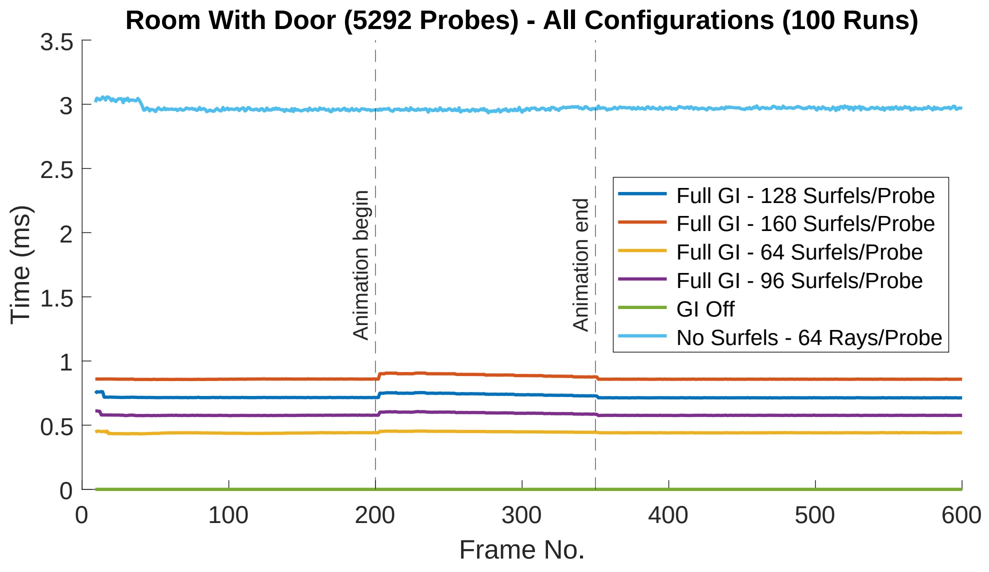


Figure 5.24: Execution times for probe shader.

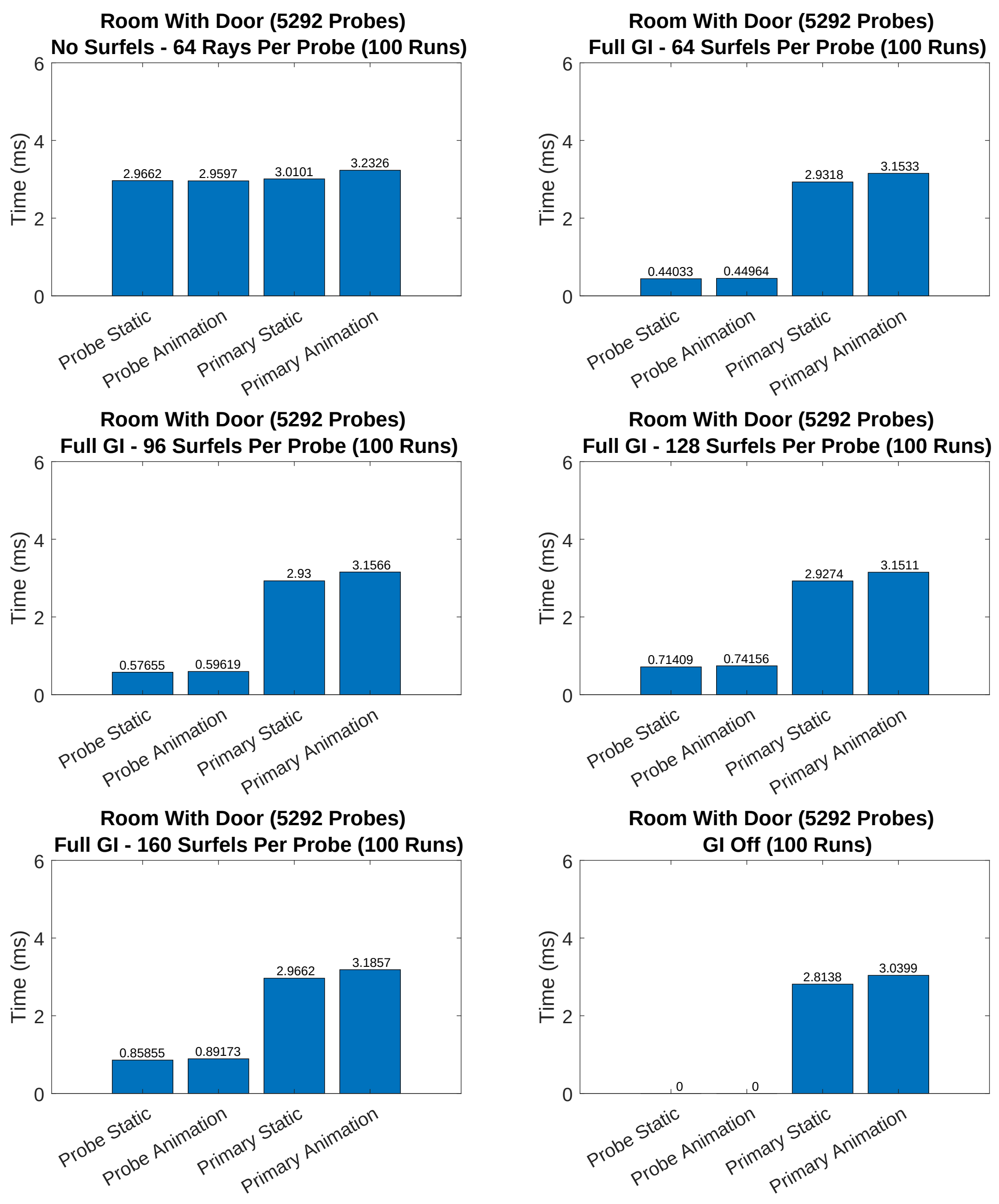


Figure 5.25: Average execution times for the probe update shader and the primary ray trace in the *Room With Door scene*, for all configurations.

5.4 Discussion

Here we discuss the results shown above. In general we can note that the DDGI system appears to work as intended, and provides global illumination to a 3D scene well within the limits for real-time frame times, for the simple scenes sampled.

5.4.1 Visual Results

The system provides DDGI as expected, but there are, of course, limitations.

Cornell Box

Figure 5.2 showcases the bounce-light achieved with the DDGI system: The sides of the blocks that are facing away from the light source are visible even though they receive no direct light, due to the light bouncing onto them from the walls, floor, and ceiling. The bounced light also carries the color of the surface it was bounced from, which is visible in the green light spilling onto the tall block and around the left wall, and the blue light spilling onto the cube and around the right wall. These are all improvements comparing to **Figure 5.1**, where all surfaces that are not directly illuminated by the light source are almost perfectly black, since they are only lit by the ambient light term.

Room With Door

Figure 5.5 shows that the light that is let in through the door opening lights up part of the floor, the right wall, and part of the back wall, and then bounces to light up the ceiling and walls. Green color spill from the right wall is also visible on the ceiling and floor, and so is the blue color spill from the back wall. Comparing to **Figure 5.4**, we see that only the parts of the room that are directly lit by the light shaft through the door are visible.

Shortcomings

We do, however, see some artifacts: One of the most obvious is the blockiness of the probe lighting, most visible on the sides of the blocks in **Figure 5.3** and in the back left corner in **Figure 5.6**. This is a natural side effect of the discretization we perform by delegating the GI to a discrete number of probes, and part of the price we pay for speeding up the lighting calculations. Thankfully, it is not very noticeable when the full lighting is turned on. We also see some spotty GI along the edges where the blocks meet the floor, due to the blocks being rotated at an angle in relation to the probe grid.

Another shortcoming is that the probes only provide one light bounce, meaning that surfaces that are not directly lit by the light source do not "give off" any probe lighting. This causes darker patches, e.g visible on the bottom of the right wall of the *Cornell Box*, where the cube blocks the light from the light source from bouncing off of the floor, without providing any other light from its own right side (since that side is facing away from the light source), even though it is itself lit by bounce light from the wall.

In the *Room With Door* scene, most noticeably in **Figure 5.6**, we see that the green light bounced from the right wall stops quite abruptly about halfway across the floor, instead of gradually fading out. This is caused by the fact that the GI light only can propagate along the diagonal of the probe grid. **Figure 5.26** illustrates this phenomenon along the xy-plane.

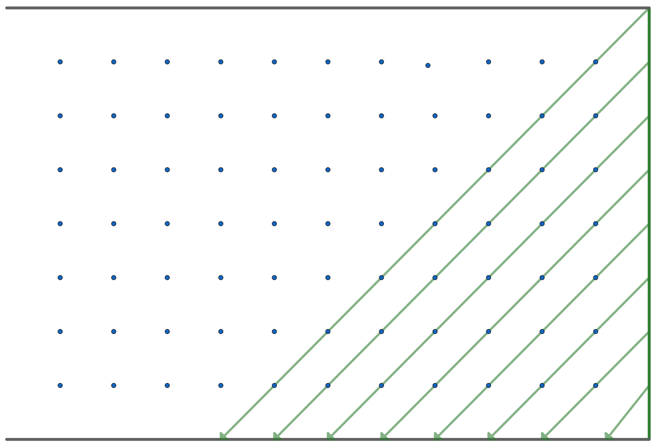


Figure 5.26: Illustration showing how the distance of the bounced light is limited by the probe grid. The light bounced off of the green wall on the right cannot reach further to the left than shown, since it can only travel diagonally along the probe grid (illustrated by the blue dots).

High Quality Shadows

The system really looks its best when combined with the ray traced soft shadows. Since the surfels also take the shadows into account when being lit, the softer transitions between light and dark also help mask over some of the artifacts of the DDGI, such as the aforementioned blockiness.

5.4.2 Execution Time Results

Probe Update Shader

As **Figures 5.11 - 5.14** and **Figures 5.19 - 5.22** in the previous section show, the probe update shader execution time is quite stable when the geometry is static, for either scene. As the animation starts the execution time goes up, since the probes start re-placing surfels, and when the animation stops it goes back to the static level. In the *Room With Door* scene, the execution time also goes down steadily during the animation, which is likely caused by the fact that the top of the door slides outside of the probe grid when it opens, and therefore causes fewer and fewer probes to re-place their surfels as it goes up. In the *Cornell Box* scene, there is a visible "dent" in the execution time, where it temporarily goes down to a stable, lower, level, before returning to the initial animation level. It is unclear why this happens.

For the *No Surfels* configuration, shown in **Figure 5.10** and **Figure 5.18**, the probe update shader execution time is practically constant, and much higher than the equivalent *Full GI - 64 Surfels Per Probe* configuration. Looking at the results of the *Full GI - 64 Surfels Per Probe* configuration and the *No Surfels - 64 Rays Per Probe* configuration in **Figure 5.17** and **Figure**

5.25, we see a very clear advantage to using the surfels. Comparing the execution times when animating (the worst case for the surfels), we see an ~ 85% decrease for the *Room With Door*, and an ~ 86% decrease for the *Cornell Box*.

For the *GI Off* configuration, shown in **Figure 5.15** and **Figure 5.23**, the probe update execution time is a constant zero, as expected.

Primary Ray Trace

The execution time for the primary ray trace is quite stable outside of the animation in both scenes, except for an initial hump at the beginning. For the *Cornell Box* scene, we see a slight dip, beginning when the animation starts and returning sharply about halfway through. This lines up with the middle of the "dent" in the probe update shader execution time mentioned before, but the reason is not clear here either.

The primary ray tracing execution times in the *Room With Door* scene drop off sharply at the tail end of the animation, and then it stays constant at this lower level when the scene is static again. This is also odd, and we do not have any definite answer as to why this happens.

For the *Cornell Box* scene, we can see that the average execution time for the primary ray tracing is almost entirely unchanged whether we are animating or not, which is what is to be expected. For the *Room With Door* scene the average execution time is slightly higher when animating, since the drop in execution time per frame after the animation has completed lowers the average.

Chapter 6
Conclusions

The results show that the system works as intended, and adds bounce lighting to 3D scenes. They also show that the performance is quite good. The caching of the surfels is shown to increase performance drastically.

6.1 Answers To Research Questions

Here we attempt to answer the research questions posed in Section 1.1 (*Research Questions*).

Is a combination of light probes and surfels together with ray tracing a good approach to achieve convincing global illumination in a dynamic scene for real-time applications?

Yes, we believe that the results show that this approach provides a good solution for global illumination, at reasonable execution times.

How much performance is gained by using surfels compared to not using them?

The results show that caching geometry information for lighting using surfels improves performance significantly. We saw an $\sim 85\%$ decrease in the execution time for the probe update shader when using surfels, compared to calculating the GI without them.

6.2 Future Work

Though we believe the system works well, there is clearly much room for improvement.

It would be a very welcome addition to the results of this report to have a comparison with some kind of ground-truth image. We originally intended to implement a full path tracer mode into the renderer to compare our results against, but time constraints got in the way. Testing with more scenes and on (many) other computers with different hardware would also strengthen the validity of our findings.

It would also be useful to do more experiments focused on assessing the scaling of the execution times as a function of the number of probes in the scene. It appears to be roughly linear, which makes sense given the construction of the system (doubling the number of probes would reasonably double the total execution time for the probe update shader), but further research is needed.

6.2.1 Suggested Areas for Further Investigation

Here follows some suggestions of strategies that might be applied to improve the performance or visual fidelity of the system.

Multiple Light Bounces

It should not be very difficult to have the surfels take the lighting from other probes into account when being lit, which would (at least in theory) allow for an infinite number of light bounces (over an infinite number of frames). We even made some attempts at this, but were unable to avoid severe feedback issues. At least part of the problem is probably the system's disregard for the conservation of energy, as discussed below.

Proper Conservation of Energy

The current solution relies in no small part on adjusting the brightness of the probe lighting with some constant factors, in order to improve the appearance of the GI lighting. This violates the conservation of energy, as we are essentially injecting energy into the system, which is not physically correct. With some investigation, it should be possible to obey the proper physical properties of light, which we believe would likely improve the visual quality and make the system more compatible with other physically based rendering solutions.

Time Amortization of Surfel Updates

As the execution time results show, the cost of lighting the probes is increased during frames where an object has moved compared to the last frame. In our testing, we moved objects continuously, updating all affected probes every frame during the whole animation. This is likely not the most clever way to go about this: If we would instead only begin to update the surfels every, say, tenth frame of an animation, we could amortize the cost of replacing the surfels over the ten coming frames, probably without much impact on the visual fidelity. For big changes to scene geometry, such as a wall crumbling, it might even make the most sense to do the update of the GI system in a single frame, instead of having the GI change continuously as the wall is gradually destroyed, and cover up the sharp change in GI with smoke and rubble.

Better Interpolation of the Probe Lighting

The final lighting contribution from the probes is calculated by simple trilinear interpolation between the eight closest probes to a given primary ray hit point. This does not obey the proper exponential behavior of light falloff, and can definitely be improved upon. Better solutions here might also improve the blocky appearance of the probe lighting.

Combine Similar Surfels

It might be possible to combine surfels from different probes that are placed close to each other on the same surface, so that we don't get a large unnecessary duplication of surfels with almost the exact same data. The best solution might even be to place surfels on geometry independently of the probes, and then have them associate to probes afterwards.

References

[1] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillarie. *Real-Time Rendering, Fourth Edition.* CRC Press, 2018.

[2] Andreas Brinck, Xiangshun Bei, Henrik Halén, Kyle Hayward. Global illumination based on surfels, 2021. `https://www.ea.com/seed/news/siggraph21-global-illumination-surfels`.

[3] Anp. Kantutjämning 2 linjer. `https://commons.wikimedia.org/wiki/File:Kantutj%C3%A4mning_2_linjer.png`.

[4] Chajdas, Matthäus G. and Weis, Andreas and Westermann, Rüdiger. Assisted environment probe placement. 2011. `https://anteru.net/research/assisted-environment-probe-placement/`.

[5] Cornell University Program of Computer Graphics. The cornell box, 1998. `http://www.graphics.cornell.edu/online/box/`.

[6] Electronic Arts Inc. Frostbite. `https://www.ea.com/frostbite`.

[7] GianniG46. File:lambert2.gif. `https://commons.wikimedia.org/wiki/File:Lambert2.gif`.

[8] Henrik. Ray trace diagram, 2008. `https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg`.

[9] Martin-Karl Lefrançois, Pascal Gautron. Dx12 raytracing tutorial, 2023. `https://developer.nvidia.com/rtx/raytracing/dxr/DX12-Raytracing-tutorial-Part-1`.

[10] Microsoft. Directx-graphics-samples. `https://github.com/microsoft/DirectX-Graphics-Samples`.

[11] Microsoft. Directx-specs. `https://microsoft.github.io/DirectX-Specs/`.

[12] Microsoft. Reference for hlsl. https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-reference.

[13] Nikolay Stefanov. Global illumination in tom clancy's the division. https://www.youtube.com/watch?v=04YUZ3bWAyg.

[14] Nvidia. Control: Multiple stunning ray-traced effects raise the bar for game graphics. https://www.nvidia.com/en-us/geforce/news/gfecnt/control-rtx-ray-tracing-dlss/.

Appendices

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2023-06-01

EXAMENSARBETE Dynamic Diffuse Global Illumination Using Probes And Surfels
Dynamisk lambertsk global belysning genom sonder och ytelement
STUDENT Elmer Dellson
HANDLEDARE Michael Doggett (LTH), Calle Lejdfors (Tencent Games, AMD)
EXAMINATOR Per Andersson (LTH)

En ny effektiv metod för att ljussätta interaktiv 3D-grafik

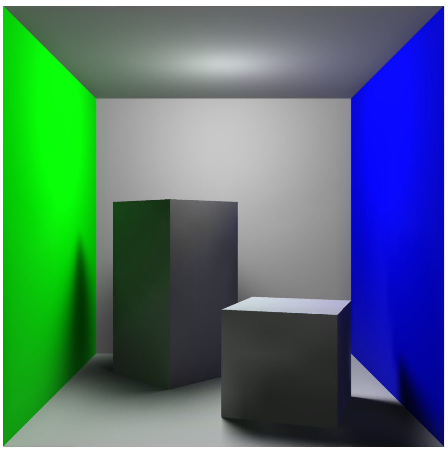
POPULÄRVETENSKAPLIG SAMMANFATTNING **Elmer Dellson**

I det här examensarbetet har vi utvecklat en ny metod för att simulera hur ljus studsar i en interaktiv 3D-värld. Metoden visade sig vara snabb och ge goda resultat.

För att en användare ska kunna interagera med 3D-världen i t.ex. ett datorspel behöver datorn generera nya bilder tiotals gånger i sekunden. För att framställa så verklighetstrogna bilder som möjligt behöver programmet simulera hur ljus beter sig. Detta är väldigt krävande, eftersom ljus består av miljardtals fotoner som rör sig i bokstavligt talat ljusets hastighet, och vi behöver därför ta till smarta genvägar för att simuleringen ska gå snabbt nog. Särskilt svårt är att beräkna hur ljus studsar och färgas av de föremål som det studsar på, och det är detta problem som vi tar oss an i examensarbetet.

Ju snabbare datorn kan framställa nya bilder, desto snabbare kan användaren interagera med programmet, vilket är viktigt när man t.ex. spelar datorspel som kräver snabba reaktioner. Om man kan snabba upp en del av bild-genereringen får datorn också tid över till att göra mer avancerade beräkningar i andra delar av programmet, och kan på så sätt framställa ännu bättre bilder.

Tekniken som utvecklades i det här examensarbetet räknar ut hur ljus studsar och flödar genom 3D-världen, och sparar den informationen strategiskt så att beräkningarna inte behöver göras om för varje ny bild. Ett system utvecklades också för att upptäcka när något i 3D-världen förändras, så att ljussättningen kan anpassas när det behövs. Många tidigare lösningar har haft begränsningen att 3D-världen inte går att ändra på medan programmet körs. Vår lösning låter däremot hela 3D-världen förändras hur som helst, när som helst.


En datorgenererad bild ljussatt av det nya systemet. Notera att ljuset från taket studsar på väggarna och färgar blocken bredvid!

Resultaten från mätningarna visar att metoden fungerar väl, och ger övertygande ljussättning väldigt snabbt. Att systemet sparar ljusets flöde mellan varje genererad bild snabbar dessutom upp beräkningarna med ca 85%. Metoden verkar vara snabb nog att kombineras med andra tekniker för att ge ännu mer verklighetstrogna bilder.